MÁSTER EN INVESTIGACIÓN EN INFORMÁTICA
Facultad de Informática. Universidad Complutense de Madrid.

PROYECTO FIN DE MÁSTER EN

SISTEMAS INTELIGENTES

2009-2010

Desarrollo Dirigido por Lenguajes de Aplicaciones de Procesamiento XML: Especificación Modular Basada en Gramáticas de Atributos y Generación Automática

Autor:

BRYAN TEMPRADO BATTAD

Director:

JOSÉ LUIS SIERRA RODRÍGUEZ

MÁSTER EN INVESTIGACIÓN EN INFORMÁTICA

PROYECTO DE MÁSTER EN

SISTEMAS INTELIGENTES

2009 / 2010

Desarrollo Dirigido por Lenguajes de Aplicaciones de Procesamiento XML: Especificación Modular Basada en Gramáticas de Atributos y Generación Automática

Autor:

BRYAN TEMPRADO BATTAD

Director:

JOSÉ LUIS SIERRA RODRÍGUEZ

FACULTAD DE INFORMÁTICA

UNIVERSIDAD COMPLUTENSE DE MADRID

Tras más quebraderos de cabeza, innumerables horas de trabajo y el esfuerzo de un año más, finalmente se ha conseguido llevar a cabo una importante y sustancial mejora del entorno XLOP.

Quiero agradecer y hacer una mención especial a José Luis por proporcionar toda la ayuda necesaria y por su entera dedicación al proyecto, y también hacer una mención especial a la contribución realizada por Alberto y Antonio durante el desarrollo de la primera versión de XLOP. A todos ellos:

-Gracias-

El/la abajo firmante, matriculado/a en el Máster en Investigación en Informática de la Facultad de Informática, autoriza a la Universidad Complutense de Madrid (UCM) a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a su autor el presente Trabajo Fin de Máster: “Desarrollo Dirigido por Lenguajes de Aplicaciones de Procesamiento XML: Especificación Modular Basada en Gramáticas de Atributos y Generación Automática”, realizado durante el curso académico 2009-2010 bajo la dirección de José Luis Sierra Rodríguez en el Departamento de Ingeniería del Software e Inteligencia Artificial, y a la Biblioteca de la UCM a depositarlo en el Archivo Institucional E-Prints Complutense con el objeto de incrementar la difusión, uso e impacto del trabajo enInternet y garantizar su preservación y acceso a largo plazo.

Bryan Temprado Battad

Resumen

En este proyecto de investigación se estudia la incorporación de mecanismos de modularidad al entorno XLOP (XML *Language-oriented processing*). XLOP es un entorno para el desarrollo de aplicaciones de procesamiento de documentos XML que permite especificar dichas aplicaciones mediante *gramáticas de atributos*, un formalismo ampliamente utilizado en la especificación de lenguajes informáticos. A partir de estas especificaciones, y de la implementación de las funciones semánticas utilizadas en las mismas, así como de la lógica específica adicional, XLOP permite generar automáticamente los programas que realizan el procesamiento. La principal ventaja de XLOP es promover una organización de las aplicaciones de procesamiento XML en dos capas bien diferenciadas: la capa *lingüística*, que media con la estructura gramatical de los documentos, y la capa de la *lógica específica de la aplicación*, que introduce el código adicional requerido, de forma que éste sea totalmente independiente de los detalles de procesamiento de dichos documentos. Así mismo, XLOP permite producir y mantener la capa lingüística utilizando especificaciones declarativas de muy alto nivel. No obstante, en la versión estable actual de XLOP, esta ventaja se ve obscurecida por su falta de modularidad. Efectivamente, la especificación de la capa lingüística se realiza mediante una única gramática monolítica, localizada en un único archivo. Este hecho dificulta el uso de XLOP durante el abordaje de tareas de procesamiento complejas. El objetivo de este proyecto es investigar en mecanismos que permitan resolver este inconveniente. Para ello, en este proyecto se propone, por una parte, mecanismos de modularización básicos, que permitan controlar los posibles conflictos de nombres que surgen en especificaciones complejas, facilitando, así mismo, la reutilización de diferentes módulos en distintas aplicaciones. Estos mecanismos están basados en un sencillo sistema de *espacios de nombres* para los símbolos no terminales de las gramáticas. Por otra parte, en el proyecto se investiga también cómo fraccionar las especificaciones en múltiples archivos, que contienen especificaciones más simples que se unen en base a las producciones de las gramáticas subyacentes. De esta forma, las especificaciones complejas se pueden modularizar en términos de múltiples *aspectos semánticos*. Llevada al límite, esta estrategia de modularización basada en aspectos semánticos puede conducir a la necesidad de disponer de múltiples sintaxis alternativas para el mismo lenguaje, cada una de las cuáles es apropiada para un determinado aspecto del procesamiento. En el proyecto se investiga cómo dar soporte a estos mecanismos flexibles de especificación que soportan *múltiples vistas sintácticas*. Como consecuencia se propone un nuevo tipo de gramáticas de atributos modulares, las *gramáticas multivista*, y se estudia cómo dar soporte operacional a dichas gramáticas mediante el algoritmo GLR, una generalización del algoritmo de análisis LR para gramáticas no LR, durante la fase de análisis, y mediante un método de evaluación de atributos que opera sobre los bosques de análisis sintácticos en los que se compactan los distintos árboles de análisis sintáctico asociados con cada vista. A fin de analizar la factibilidad práctica de estas propuestas, en este proyecto de investigación se han implementado también la mayor parte de los aspectos de las citadas extensiones en el entorno XLOP, haciendo especial énfasis en los aspectos operacionales: análisis basado en el método GLR y evaluación de atributos orientados a los bosques de análisis sintáctico producidos por dicho método.

Palabras clave: XML, Procesamiento de Documentos XML, Gramática de Atributos Modular, Desarrollo Dirigido por Lenguajes, Procesador de Lenguaje, Análisis GLR, Bosques de Análisis Sintáctico, Evaluación de Atributos, e-Learning, Sistema Tutor.

Abstract

This research project addresses the enhancement of the XLOP environment (XML *Language-Oriented Processing*) with modularization mechanisms. XLOP is an environment for the development of XML-processing applications that makes it possible to specify these applications by means of *attribute grammars*, a formalism widely used in the specification of computer languages. Using these specifications, and with the additional support of the implementation of the semantic functions and the application-specific logic, XLOP makes possible the automatic generation of the processing programs. The main advantage of XLOP is to promote an organization of the processing applications in two clearly-bounded layers: the *linguistic* layer, which is in charge of mediating with the grammatical structure of the documents, and the *application-specific logic*, which introduces the additional code required without compromising it with specific XML processing details. Moreover, XLOP allows the production and maintenance of the linguistic layer using very high-level declarative specifications. However, in the current stable version of XLOP, this advantage is affected by its lack of modularity. Indeed, the specification of the linguistic layer is carried out with a single monolithic grammar, located in a single archive. It hinders the use of XLOP for complex XML processing tasks. The goal of this project is to solve this drawback. For this purpose, we propose, on one hand, basic modularization mechanisms for controlling the potential name crashes arising in complex specification. These mechanisms, which are based on namespaces for non-terminal symbols, facilitate module reuse in different applications. On other hand, the project also inquiries on how to split the specifications in multiple files, each containing simpler specifications that can be joined in terms of the syntax rules of the underlying grammars. It makes possible the modularization of complex specifications in terms of different *semantic aspects*. In the limit, this modularization strategy can lead to propose multiple alternative syntaxes for the same language, each being suitable for a particular processing aspect. In this project, we investigate how to provide flexible specification mechanisms for supporting multiple *syntactic views*. Therefore, we propose a new type of modular attribute grammars, the *multiview grammars*, and we study how to give operational support for these grammars. Our solution is based on the GLR algorithm, a generalization of the LR parsing algorithm for non-LR grammars, for the analysis of the documents. For evaluating the attributes, our proposal operates on the parse forest in which the parse trees associated with each syntactic view are compacted. In order to analyze the practical feasibility of these proposals, we have implemented the most of the aforementioned extensions in the XLOP environments, and, in particular, the cited operational aspects: GLR parsing and parse-forest attribute evaluation.

Keywords: XML, XML Processing, Modular Attribute Grammar, Language-driven Development, Language Processor, GLR Parsing, Parse Forest, Attribute Evaluation, e-Learning, Tutoring System.

Índice

Capítulo 1
Prólogo

1.1 Introducción

El problema abordado en este proyecto de investigación es el del *procesamiento* de documentos XML. XML (eXtensible Markup Language) [Bray et al. 2008], especificación propuesta a finales de los 90 por el Word Wide Web Consortium (W3C) como evolución del estándar SGML (Standard Generalized Markup Language) [Goldfarb 1991], es un estándar *de hecho* utilizado para la representación de *documentos electrónicos*. La importancia del lenguaje radica en que la información intercambiada entre los distintos componentes de un sistema informático puede entenderse en muchas ocasiones como *documentos electrónicos*. Debido a este hecho, XML es una tecnología esencial en cualquier escenario moderno de desarrollo de *software* o de gestión de la información.

XML es un lenguaje informático que norma cómo añadir a los contenidos de un documento electrónico la *metainformación* necesaria para explicitar su estructura (p.ej., su *título*, sus *autores*, sus *capítulos*, cada *sección* en cada capítulo, etc.). Dicha metainformación consiste en un conjunto de *marcas* o *etiquetas* debidamente anidadas, que describen la estructura del documento y que, por tanto, facilitan el posterior (o posteriores) procesamiento(s) del mismo. Además de introducir un criterio estándar para marcar documentos, XML también establece mecanismos que permiten restringir los posibles usos del marcado (p.ej., en el interior de la lista de autores no puede aparecer un capítulo). Estos mecanismos se rigen por un modelo *lingüístico*: para acotar los posibles usos del marcado debe especificarse una *gramática formal*. XML incluye un sub-lenguaje para describir tales gramáticas (el sub-lenguaje de las DTDs – *Document Type Definitions*). Así mismo, la comunidad XML ha propuesto también otras muchas alternativas para describir tales *gramáticas documentales*, que extienden o complementan las capacidades expresivas de las DTDs [Murata et al. 2005].

XML, sin embargo, *no* norma cómo deben procesarse los documentos en una determinada aplicación informática. De hecho, esta separación entre estructura y procesamiento es uno de los principios básicos de XML: el marcado en XML es *descriptivo*, orientado a plasmar la estructura lógica de los documentos, pero no las posibles formas de procesar dichos documentos [Coombs et al. 1987]. El procesamiento de los documentos transciende el propósito de XML, y debe llevarse a cabo utilizando otros medios. Este hecho desemboca, por tanto, en el problema en el que se enmarca este proyecto: *cómo llevar a cabo el procesamiento de los documentos* XML.

Desde la aparición de XML se han propuesto también múltiples tecnologías para abordar el problema del procesamiento de los documentos XML. Dichas tecnologías pueden clasificarse, inicialmente, en *tecnologías específicas* y *tecnologías de propósito general*:

- Las tecnologías específicas se centran en un tipo específico de tareas (p.ej., *consulta* de la información contenida en el documento, o *transformación* del documento a otro tipo de formato). De esta forma, cada una de estas tecnologías es aplicable a dicho tipo específico de tareas (p.ej., XSLT es un lenguaje que permite especificar transformaciones de documentos [Kay 2007]).

- Las tecnologías de propósito general son aplicables a cualquier tarea de procesamiento de documentos XML. Estas tecnologías se presentan, normalmente, como un conjunto de herramientas o como un marco de aplicación embebidos en un lenguaje de programación de propósito general (p.ej., Java, PHP o C#). El núcleo de una tecnología de este tipo es un analizador o parser XML: un artefacto software que lee documentos XML, comprueba el correcto uso del marcado, y ofrece una interfaz adecuada a los elementos de información contenidos en los documentos [Lam et al. 2008]. Utilizando, entonces, el lenguaje de programación genérico en el que se embebe la tecnología, se programa el procesamiento deseado: el resultado de esta actividad de programación es la lógica específica de la aplicación. Dicha lógica específica accede a la información de los documentos a través de la interfaz ofrecida por el parser, y lleva a cabo el procesamiento requerido.

Independientemente de su naturaleza, todas las tecnologías descritas comparten una característica común: el entender los documentos XML como *estructuras de datos*. Esta visión orientada a los datos contrasta, sin embargo, con la concepción lingüística original de las aplicaciones XML. Efectivamente, tal y como se ha indicado anteriormente, formular una aplicación XML es equivalente a proponer un lenguaje de marcado específico para un determinado tipo de documentos, lenguaje que viene definido mediante un modelo lingüístico formal: la gramática documental. Este hecho conduce, por tanto, a una reflexión evidente: si el objeto de procesamiento en una aplicación XML es un lenguaje, ¿*por qué no entender dicha aplicación como un procesador de lenguaje*, *y de esta forma aprovechar el enorme arsenal de conocimiento, métodos, técnicas y herramientas disponible en un dominio tan maduro como es el de la construcción de procesadores de lenguaje?* (véase, por ejemplo, [Aho et al. 2007]). En este proyecto de investigación se explora esta posibilidad.

1.2 Contexto y Objetivos del proyecto

Este proyecto de máster se encuadra dentro de una línea de investigación en el procesamiento orientado a lenguajes de documentos XML llevada a cabo en el Grupo de Investigación en Ingeniería del Software y e-Learning del Dpto. de Ingeniería del Software e Inteligencia Artificial de la Universidad Complutense de Madrid. El proyecto en sí parte de los

resultados previos obtenidos con la construcción del sistema XLOP (XML Language Oriented Processing) [Martínez & Temprado 2009]. XLOP es un entorno de desarrollo que permite *entender* la construcción de aplicaciones XML como la construcción de procesadores de lenguaje. Más concretamente, el entorno permite:

- Especificar cada aplicación de procesamiento XML como un procesador para el lenguaje de marcado definido por su gramática documental. Para ello el entorno ofrece un lenguaje de especificación basado en *gramáticas de atributos* [Knuth 1968; Paaki 1995].

- Generar automáticamente el procesador a partir de dicha especificación.

La principal ventaja de XLOP es, de esta forma, permitir desarrollar aplicaciones de procesamiento XML a un nivel mucho más alto de abstracción que las actuales propuestas orientadas a datos. No obstante, la actual versión estable de XLOP (véase [Martínez & Temprado 2009] y [Sarasa et al. 2009d] para una descripción detallada) exhibe también una desventaja fundamental: su falta de modularidad. Efectivamente, en dicha versión estable, las especificaciones se reducen a gramáticas monolíticas, en las que cada regla amalgama la parte sintáctica con todas y cada una de las ecuaciones semánticas involucradas. El resultado es que, para aplicaciones de procesamiento complejas, las especificaciones resultantes pueden ser difíciles de comprender y mantener.

El objetivo general de este proyecto es, por tanto, buscar mecanismos que permitan dotar de modularidad a especificaciones de procesamiento de documentos XML basadas en gramáticas de atributos, así como formas de generar automáticamente los programas de procesamiento a partir de dichas especificaciones. Este objetivo general se concreta en los siguientes objetivos específicos:

- Permitir fraccionar especificaciones complejas en fragmentos más simples, cada uno de los cuales pueda albergarse en un archivo distinto. Con el fin de controlar las posibles colisiones de nombres que pueden surgir en este proceso, se introducirá un mecanismo de espacios de nombres que permita compartimentar los símbolos utilizados en las gramáticas (más concretamente, los no terminales).

- Permitir descomponer especificaciones de procesamiento complejas en aspectos semánticos más simples y manejables. Dichos aspectos podrán, posteriormente, componerse para producir las especificaciones finales.

- Encontrar mecanismos que permitan armonizar, en una misma especificación, distintas *vistas sintácticas* de un mismo lenguaje de marcado. Efectivamente, la descomposición de una tarea de procesamiento de un tipo de documentos XML en aspectos más simples puede producir sub-tareas que requieran estructurar el mismo fragmento de documento de maneras distintas. Como resultado, en la misma especificación deberán coexistir distintas gramáticas incontextuales para el mismo lenguaje de marcado. Se necesitarán definir, por tanto, formas de asegurar

la coherencia de estas gramáticas entre sí, así como con la DTD o esquema que define el lenguaje de marcado.

- Encontrar formas de definir modularmente gramáticas de atributos sobre múltiples vistas sintácticas de un mismo lenguaje de marcado. Dichas definiciones deberán contemplar, así mismo, la operación coordinada de los procesamientos dirigidos por las distintas vistas.

- Definir un modelo genérico de ejecución para especificaciones que cumplan las características aludidas. Dicho modelo servirá como base para transformar automáticamente dichas especificaciones en programas de procesamiento de documentos XML ejecutables.

- Extender XLOP con distintas características que faciliten la especificación modular de tareas de procesamiento de documentos XML, y que soporten (al menos parcialmente) los mecanismos de especificación y ejecución a los que se ha hecho alusión en los puntos anteriores.

Como objetivo adicional, este trabajo persigue también mostrar el potencial y el uso de las técnicas de procesamiento de documentos XML dirigido por lenguajes (y, en particular, de XLOP) en el desarrollo de programas de procesamiento de documentos XML, centrándose principalmente en la construcción de generadores de aplicaciones que operan guiados por documentos XML. En particular, como caso de estudio se utilizará una aplicación no trivial en el dominio de e-Learning: <e-Tutor>, una aplicación para la generación de *sistemas tutores* [Sleeman & Brown 1986] a partir de documentos XML.

1.3 Estructura del documento

La estructura de esta memoria es la siguiente:

- En el Capítulo 2 se realiza un estudio de los conceptos y tecnologías más relevantes e importantes para este trabajo. En particular, en este capítulo se realiza un análisis detallado del algoritmo GLR, una generalización del algoritmo de análisis sintáctico LR que permite tratar con tablas de análisis con múltiples entradas, ya que el núcleo básico de la propuesta realizada en este proyecto se basa en dicho algoritmo.

- En el Capítulo 3 se presenta la versión 1.0 del entorno XLOP junto a las experiencias de uso de dicho entorno para la creación de aplicaciones de procesamiento XML. En este capítulo se presenta una extensión del entorno para el soporte de especificaciones modulares y se introduce el concepto de gramáticas de atributos multivista, una extensión de las gramáticas de atributos convencionales que soporta múltiples vistas sintácticas para un mismo lenguaje.

- En el Capítulo 4 se introduce el modelo de evaluación para gramáticas de atributos multivista y se detalla su implementación en el entorno XLOP.
- Por último, el Capítulo 5 presenta las principales conclusiones obtenidas con el desarrollo de este trabajo, así como propone posibles extensiones y trabajos futuros.

La bibliografía referida en el documento se incluye como última sección del mismo.

Capítulo 2
Estado de la Cuestión

2.1 Introducción

En este capítulo se presenta una introducción a los aspectos conceptuales y tecnológicos más importantes que tienen relación directa con este proyecto de máster. Primero, se presenta una breve introducción a los lenguajes de marcado, destacando XML como el lenguaje de marcado más utilizado en la actualidad, y que será el tratado por nuestro proyecto, así como se introducen los tipos de marcos genéricos para el procesamiento de los documentos XML más ampliamente utilizados actualmente. Seguidamente, se analiza con detalle uno de los componentes fundamentales en el que se apoyará el modelo de ejecución para programas de procesamiento de documentos XML generados a partir de especificaciones modulares basadas en gramáticas de atributos: el método de análisis sintáctico GLR. Efectivamente, al ser capaz de tratar eficientemente tipos suficientemente amplios de gramáticas ambiguas, este componente será esencial para soportar especificaciones de procesamiento de documentos XML que integran múltiples vistas sintácticas, cada una de ellas caracterizada mediante una gramática incontextual sobre un tipo de documento o subdocumento común. Por último, se introducen algunos elementos relativos al formalismo de gramáticas de atributos, a los mecanismos de modularización propuestos en relación con dicho formalismo, y a los métodos de evaluación dinámica de atributos, al ser estos últimos una base igualmente esencial en nuestra propuesta de modelo de ejecución.

2.2 Lenguajes de marcado y XML

2.2.1 Lenguajes de marcado

Los lenguajes de marcado permiten codificar un documento de manera que, junto con el texto, se incorporan etiquetas o marcas que aportan información adicional acerca de la estructura del documento o de su presentación [Coombs et al. 1987]. La idea de marcado siempre ha estado muy relacionada con la presentación de la información. A lo largo del tiempo, los procesadores de texto han ido evolucionando para permitir presentaciones más sofisticadas y elegantes. Estos procesadores introducen marcado en sus documentos para indicar qué tipo de acción deben realizar con el texto que sigue (p.ej., un cambio de fuente o un cambio de línea). Sin embargo, estas marcas o etiquetas quedan ocultas al usuario del procesador de texto, que sólo ve el texto formateado. Hoy en día, a los procesadores de texto se les pide mucho más que mostrar textos con distintas fuentes. Estas nuevas necesidades han

hecho que los procesadores de texto cambien el uso que dan al marcado, pasando a utilizarlo para realizar cada vez procesos más complejos.

Los lenguajes de marcado son de diferente naturaleza a los lenguajes de programación. Mientras que los lenguajes de programación están orientados a describir programas informáticos, y son utilizados por programadores, los lenguajes de marcado se han utilizado tradicionalmente, y se siguen utilizando, en la industria editorial y de la comunicación. Sus usuarios no son expertos en informática, sino que son autores, editores e impresores.

Dependiendo de la funcionalidad dentro de un documento, el marcado se puede dividir en varios tipos [Coombs et al. 1987]. Cada uno de estos tipos de marcado presenta ventajas y desventajas desde el punto de vista del intercambio de información.

2.2.1.1 Marcado presentacional y puntuacional

El marcado presentacional es útil para aumentar la legibilidad de un documento. Ejemplos típicos de marcado presentacional son los espacios en blanco y el indentado. Este tipo de marcado aumenta la legibilidad de los datos, pero resulta insuficiente para soportar el procesamiento automático de la información. Sirva como ejemplo, “Estoesuntexto” y “Esto es un texto”: aunque ambos textos contienen la misma información, el segundo es más legible. Los espacios en blanco son marcas presentacionales que aumentan la legibilidad sin cambiar el significado de los contenidos.

El marcado de puntuación ayuda a la correcta interpretación de los contenidos. Por ejemplo, la interpretación de “Esto es un texto” cambia si se introducen los símbolos de interrogación: “¿Esto es un texto?”. Los marcados de presentación y de puntuación están totalmente extendidos en la creación de documentos, hasta el punto de que se pueden considerar intrínsecamente unidos a los contenidos. Esto hace que también se conozca a estos tipos de marcado como marcados implícitos.

2.2.1.2 Marcado procedimental

El marcado procedimental consiste en marcas que establecen la realización de unas determinadas acciones sobre los contenidos. Está orientado a realizar presentaciones del texto sobre diferentes formatos (por ejemplo, impresión en papel), está formalizado y perfectamente distinguido de los contenidos. Debido a que el programa que procesa un documento marcado procedimentalmente debe interpretar el código en el mismo orden en el que aparece, para formatear un texto, debe existir una serie de órdenes inmediatamente antes del texto en cuestión. Esto indicará los efectos a aplicar sobre el texto. Inmediatamente, después del texto deberán existir órdenes inversas que reviertan dichos efectos. Por ejemplo, en los procesadores de texto se puede indicar que un texto debe mostrarse en **negrita**, en *cursiva*, o con un tamaño de fuente más grande que el habitual. En sistemas más avanzados

se utilizan macros o pilas que facilitan el trabajo. Algunos ejemplos de sistemas de marcado procedimiental son troff [Emerson 1986], TeX [Seroul 2008], y PostScript [Adobe 2007].

2.2.1.3 Marcado descriptivo

El marcado descriptivo permite realizar una separación entre la estructura lógica de los documentos y su procesamiento, a fin de resolver las desventajas que presenta el marcado procedimental. Las etiquetas se utilizan para describir los fragmentos de texto sin especificar la manera en que estos han de ser finalmente presentados. Este tipo de marcado aparece, por ejemplo, al indicar que un determinado fragmento de texto en un libro es un título, un párrafo, o el nombre de un autor. El marcado descriptivo es flexible debido a que permite realizar cambios sin alterar el esquema de marcado. Es sencillo, debido a que simplifica la tarea de reformatear un texto, puesto que la información del formato está separada del propio contenido. Así mismo, presenta un carácter atributivo, al reflejar la estructura lógica de los contenidos en lugar de especificar la forma de procesarlos. Esto permite tener claramente identificados a los integrantes de las distintas categorías lógicas del documento [Goldfarb 1991]. Los lenguajes expresamente diseñados para generar marcado descriptivo son SGML (S*tandard* G*eneralized* M*arkup* L*anguage*) [Goldfarb 1991] y XML (*eXtensible* M*arkup* L*anguage*) [Bray et al. 2008].

2.2.1.4 Marcado referencial y metamarcado

El marcado referencial permite referir entidades que almacenan contenidos, a su vez, posiblemente marcados. Este tipo de marcado surge para facilitar el mantenimiento de documentos complejos, permitiendo por ejemplo, referir fragmentos de un documento en un documento maestro.

El metamarcado permite definir el uso y aplicación de otro marcado. Este tipo de marcado permite definir las reglas de combinación que debe seguir cierto sistema de marcado descriptivo (por ejemplo, dentro de un título no se admite un párrafo), definir los referentes válidos utilizados posteriormente en el marcado referencial, o abstraer procedimientos que, posteriormente, se utilizan para procesar contenidos en un sistema procedimiental.

2.2.2 XML

Como se ha indicado anteriormente, XML son las siglas de Extensible Markup Language, un lenguaje de marcado desarrollado por el World Wide Web Consortium (W3C) para el marcado de contenidos en Internet. Es una simplificación y adaptación de SGML orientada a facilitar el uso mediante la eliminación de las características más específicas y problemáticas que éste último presenta. XML se propone como un estándar para el intercambio de información estructurada entre diferentes plataformas. No es realmente un lenguaje en particular, sino que

permite definir la gramática de lenguajes de marcado específicos, así como establece las reglas genéricas de aplicación del marcado a los documentos. XML es una tecnología sencilla que permite compartir la información entre sistemas de una manera fácil, segura y fiable. Aparte de la propia especificación [Bray et al. 2008], algunas referencias útiles sobre XML son [Bradley 2001], [Birbeck et al. 2001] y [Maruyama et al. 2002].

2.2.2.1 Estructura de un documento XML

La tecnología XML pretende establecer una manera estructurada, abstracta y lo más reutilizable posible para expresar la información. La información se mantiene estructurada al componerse de conjuntos bien definidos de información, que a su vez, pueden componerse de subconjuntos de información. De esta manera se obtiene la información estructurada en forma de árbol a través de las marcas de etiquetas que definen fragmentos de información con un sentido claro y definido. XML surge en 1998 como fruto del consorcio W3C como metalenguaje para el intercambio de información en internet. Su desarrollo se basa en la simplificación del SGML extrayendo un subconjunto de especificaciones que aseguran la mayoría de las cualidades del metalenguaje con la menor complejidad posible. En la actualidad, XML se ha convertido en un estándar para el intercambio de información entre aplicaciones que permite asegurar la validez de los documentos mediante la incorporación explícita de una DTD (Document Type Definition).

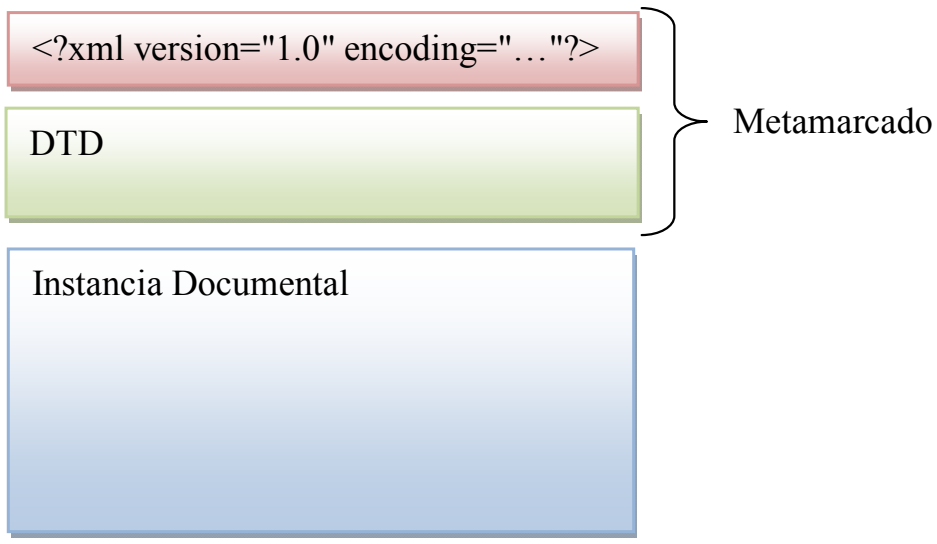


Figura 2.2.1. Estructura de un documento XML.

En la Figura 2.2.1 se presenta la estructura de un documento XML. En ella se pueden diferenciar claramente tres partes:

- El encabezado del documento se compone de metainformación que va dirigida a los procesadores XML y no se considera parte de la información del documento.

Al comienzo del documento se declara la versión XML y el conjunto de caracteres utilizado. Más concretamente, XML obliga a los sistemas de procesamiento a soportar como mínimo, Unicode en sus codificaciones UTF-8 y UTF-16 [Bradley 2001].

- La estructura del documento se especifica a través de una DTD. Las DTDs en XML son una simplificación de las DTDs SGML y permiten comprobar y validar la estructura de los documentos XML.

- La instancia documental contiene la información del documento, organizada en una estructura jerárquica mediante el uso de elementos o partes de información contenidas entre etiquetas de apertura y cierre.

La Figura 2.2.2 muestra un ejemplo de documento XML con cada una de las tres partes distinguidas en la Figura 2.2.1.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!DOCTYPE Recetas SYSTEM "Recetas.dtd" [<!ELEMENT Recetas (Receta)*>]>

<Recetas>
     <Receta tipo=”carnes”>
          <Plato>Nombre del plato</Plato>

          <Ingredientes>
               <Ingrediente>Ingrediente 1</Ingrediente>
               <Ingrediente>Ingrediente 2</Ingrediente>
          </Ingredientes>

          <Pasos>
               <Paso>Primer paso a realizar</Paso>
               <Paso>Segundo paso a realizar</Paso>
               <Paso>Tercer paso a realizar</Paso>
          </Pasos>
     </Receta>
</Recetas>
```

Figura 2.2.2. Ejemplo de documento XML.

Los documentos XML organizan la información en unidades o *elementos* que dividen la información en categorías lógicas. Los elementos son información contenida entre marcas establecidas por una etiqueta de apertura y su correspondiente etiqueta de cierre (por ejemplo, <Recetas> y </Recetas> de la Figura 2.2.2). En XML, al igual que en SGML, es posible asociar con los elementos pares *atributo-valor* para indicar metainformación adicional asociada con los mismos, o para expresar relaciones adicionales entre elementos. La etiqueta de apertura <Receta> de la Figura 2.2.2 muestra un ejemplo de *atributo-valor* para indicar información adicional sobre el tipo de una receta.

Mediante una DTD se establecen los elementos o las reglas que deben cumplir los documentos XML para garantizar que presentan la misma estructura lógica.

```
<?xml version="1.0" encoding="ISO-8859-1" ?>

<!ELEMENT Receta (Plato, Ingredientes, Pasos)>
        <!ELEMENT Plato (#PCDATA)>
        <!ATTLIST Plato tipo CDATA #IMPLIED>

      <!ELEMENT Ingredientes (Ingrediente)>
              <!ELEMENT Ingrediente (#PCDATA)>

      <!ELEMENT Pasos (Paso)>
               <!ELEMENT Paso (#PCDATA)>
```

Figura 2.2.3. Ejemplo de DTD de la Figura 2.2.2.

Estos archivos DTD permiten garantizar que los documentos XML siguen una estructura y formato especifico y válido para asegurar su correcto procesamiento. La Figura 2.2.3 muestra la DTD que presenta la estructura lógica de las instancias XML para cada receta.

Es interesante conocer que XML permite disponer de documentos que no incluyen una DTD. Sin embargo, dichos documentos sin DTD deberán seguir la estructura que el procesador de documentos XML requiere, es decir, deberán estar “bien formados”. Los documentos denominados como "bien formados" son aquellos que cumplen con todas las definiciones básicas de formato (siendo la más importante el correcto anidamiento de las etiquetas) y pueden, por lo tanto, analizarse correctamente con cualquier analizador sintáctico que cumpla con la norma. La caracterización gramatical de lenguajes basados en XML, mediante DTDs u otros mecanismos de definición gramatical de lenguajes de marcado más avanzados [Murata et al. 2005], es esencial, no obstante, de cara a especificar de forma clara y precisa la estructura del tipo de documentos en un determinado dominio, así como uno de los elementos distintivos del marcado descriptivo [Goldfarb 1981].

En el Capítulo 3 se muestra la construcción de ejemplos de aplicaciones en XLOP junto a la especificación de los documentos XML y la importancia que implica el disponer de DTDs para diseñar y llevar a cabo el correcto procesamiento de dichos documentos.

2.2.3 Procesamiento de Documentos XML

Siguiendo las directrices del marcado descriptivo [Coombs et al. 1987], XML no norma la forma de procesar los documentos marcados. Para llevar a cabo el procesamiento es necesario, por tanto, otros medios. Para ello pueden emplearse lenguajes específicos, que abordan algunos tipos de procesamiento. Por ejemplo, XSLT (*eXtensible Stylesheet Language Transformations*) [Kay 2007] es un lenguaje orientado a la transformación de documentos XML en documentos XML, que constituye un buen ejemplo de esta tendencia.

No obstante, para la realización de tareas más complejas y abiertas, es necesario utilizar marcos de procesamiento XML más genéricos [Lam et al. 2008]. Estos marcos ofrecen típicamente funcionalidades básicas para realizar el procesamiento. Entre estas funcionalidades destacan funcionalidades de lectura (a través de componentes denominados

parsers XML), comprobación de buena formación, y validación de documentos XML, así como exposición de los documentos leídos en un formato apropiado para su procesamiento. Los tipos más usuales de marcos de procesamiento genéricos para XML son los marcos *orientados a árboles*, y los *orientados a eventos*.

Los marcos orientados a árboles promueven la creación de estructuras arborescentes en base al contenido del documento. Estos marcos incluyen *parsers* capaces de transformar los documentos en árboles. Los árboles permiten el acceso inmediato a cualquier parte de la información en cualquier momento, al contrario de lo que sucede en los marcos orientados a eventos, donde el acceso a la información es secuencial. El alojamiento de toda la información en memoria permite realizar cálculos complejos, así como usar programación recursiva, a diferencia de los marcos orientados a eventos, donde suele ser preciso guardar la información en variables o incluso realizar dos pasadas sobre el documento. Sin embargo, el alojamiento de toda la información para documentos muy grandes implica un elevado consumo de recursos y mayor tiempo de proceso en la construcción del árbol, por lo que son menos eficientes en tiempo y espacio que los marcos orientados a eventos.

Los marcos orientados a eventos recorren los documentos de manera secuencial disparando un evento cada vez que se reconoce un componente. Para el uso de este tipo de marcos sólo es necesario codificar los tratamientos de eventos que se disparan al reconocer un tipo determinado de componente y realizar la gestión de la información de dicho componente, información que se obtiene como parámetro del evento correspondiente. La principal ventaja de los marcos orientados a eventos es que son sencillos de programar y rápidos en procesamiento, debido a que sólo disparan eventos al ir recorriendo el documento. Al no almacenar la información en estructuras internas, requieren pocos recursos durante el procesamiento. Esto hace interesante su utilización para procesar grandes documentos. Sin embargo, para procesamientos complejos, puede ser necesario mantener la información en una estructura de rápido acceso, objetivo que no cumplen estos marcos.

De esta forma, la elección de un tipo u otro de marco radica en el proceso a realizar y el tamaño de los documentos a tratar. Si el proceso es complejo y el tamaño de los documentos no es elevado, el uso de un marco orientado a árboles puede ser mucho más interesante que el uso de un marco orientado a eventos. Un compromiso intermedio es el proporcionado por los denominados *marcos* pull, de los cuáles StAX es la especificación más significativa [Lam et al. 2008].

Por otra parte, otra de las características que deben tenerse en cuenta es que los modelos de información en los que se basan este tipo de marcos abordan la representación de *cualquier* tipo de documento XML, independientemente de la naturaleza específica de la aplicación. Así, por ejemplo, los árboles en un marco orientado a árboles se construirán mediante objetos del tipo *Elemento*, *Contenido*, *Atributo*, etc., en lugar de en términos de conceptos semánticos propios de cada aplicación particular (p.ej., *Factura*, *Libro*, *Rectángulo*, etc.). Esta genericidad, aunque conveniente para abordar el procesamiento de cualquier tipo de documento, redunda en una mayor dificultad de programación de los programas de

procesamiento, así como de mantenimiento de los mismos. Las propuestas de *vinculación de datos* [Birbeck et al. 2001] tratan de paliar esta desventaja, permitiendo generar automáticamente representaciones específicas para cada lenguaje basado en XML, mediante la transformación de la DTD o esquema documental en un conjunto de clases orientadas a representar, en términos del dominio de aplicación, los documentos conformes con dicho lenguaje. A continuación se examina brevemente las características de los marcos de procesamiento XML más ampliamente utilizados: DOM y SAX.

2.2.3.1 Un marco orientado a árboles: DOM

Es importante disponer de algún mecanismo estándar para acceder a los árboles construidos por los marcos orientados a árboles a fin de poder independizar la lógica de la aplicación del tipo de marco utilizado. DOM es el estándar más comúnmente utilizado para la creación y modificación de estos árboles documentales [DOM 2009].

DOM son las siglas de *Document Object Model*. Ha surgido como fruto de los esfuerzos dirigidos por el consorcio W3C. Su especificación aporta un conjunto de interfaces estándar para la construcción, acceso y modificación dinámica de contenido estructurado en árboles. Se caracteriza por ser esencialmente una interfaz de programación de aplicaciones que proporciona un conjunto estándar de objetos para representar documentos HTML y XML, un modelo estándar sobre cómo pueden combinarse dichos objetos, y una interfaz estándar para acceder a ellos y manipularlos. DOM está pensado para su uso en lenguajes orientados a objetos como C++ o Java. El modelo estructural que establece para árboles de documentos XML no indica cómo dichos árboles deben ser físicamente representados.

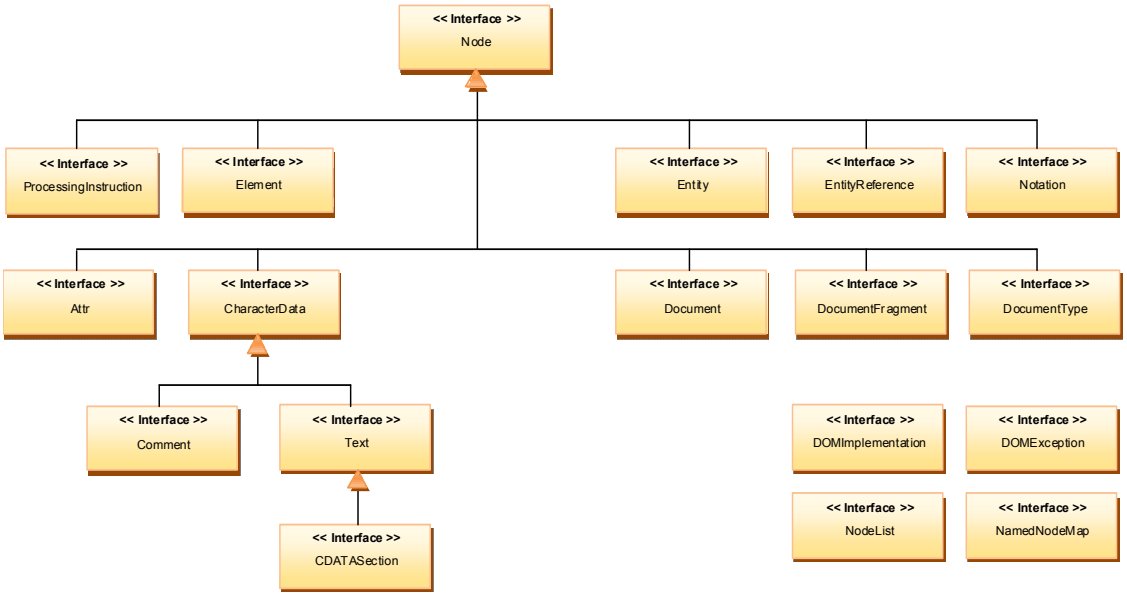


Figura 2.2.4. Diagrama de clases de la estructura DOM de nivel 1.

Existen varias especificaciones de la arquitectura clasificada en niveles con el fin de integrar facilidades adicionales. Mientras que el nivel 1 se limita a capturar el modelo estructural básico introducido por la especificación, el nivel 2 en DOM introduce características tales como espacios de nombre o propagación de eventos. También se ha incorporado un nivel 3 en DOM que hace uso de la DTD y la validación de documentos.

La Figura 2.2.4 muestra la arquitectura DOM de nivel 1, que caracteriza la estructura básica de los árboles documentales. La estructura de DOM consta de árboles de nodos cuyo tipo depende del contenido. Los nodos pueden ser Document, Element, Attr, Text, Comment, etc., tal y como muestra la Figura 2.2.4. A modo de ejemplo, los nodos de tipo Text llevan asociados como valor el contenido del texto, mientras que los nodos de tipo Element llevan asociados una secuencia de nodos hijos y una tabla de atributos. Esta interfaz proporciona un conjunto de métodos específicos que permiten acceder y modificar los valores de los atributos del elemento. Cabe destacar la interfaz Document, que centraliza la creación de los diferentes nodos mediante métodos factoría que permiten realizar implementaciones concretas sobre el resto de los elementos, así como especializar adecuadamente una implementación de DOM. Mediante estas interfaces, se establece una relación entre los nodos que permite recorrer la estructura iterando sobre la secuencia de los nodos hijos de los elementos, o bien aprovechando las relaciones especiales entre nodos.

En resumen, DOM proporciona una interfaz estándar que permite independizar las aplicaciones de las representaciones concretas de los documentos, así como conectar fácilmente distintas aplicaciones entre sí.

2.2.3.2 Un marco orientado a eventos: SAX

SAX son las siglas de S*imple* API *for* XML. SAX es la arquitectura estándar para la lectura y procesamiento de documentos XML siguiendo el modelo orientado a eventos [Brownell 2002]. Aunque su desarrollo no ha sido controlado por el W3C, ha contado con su respaldo. SAX existe en diferentes versiones. La versión 1.0 se propuso en 1998 y se mejoró con inclusión de diversas extensiones, como el soporte para espacios de nombres, en la versión 2.0 en el 2002. SAX define un API orientado a objetos que requiere la configuración específica del tratamiento de eventos asociados al procesamiento XML.

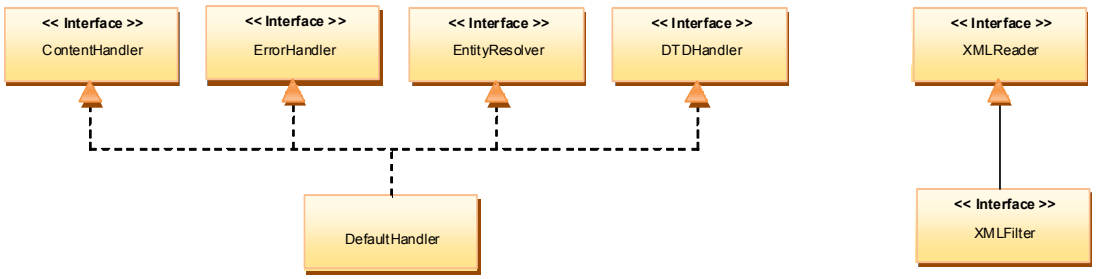


Figura 2.2.5. Diagrama de clases de la estructura SAX.

La Figura 2.2.5 muestra las diferentes interfaces que permiten gestionar los eventos clasificados según el tipo de evento que disparan. Más concretamente:

- DocumentHandler: Se caracteriza por ser la principal interfaz, cuyos métodos se invocan a medida que se reconocen los distintos tipos de nodos en el procesamiento del documento.
- DTDHandler: Interfaz que dispone de métodos que se invocan para gestionar los eventos relacionados con la DTD.
- ErrorHandler: Interfaz cuyos métodos se invocan para notificar diferentes tipos de errores durante el procesamiento del documento.
- EntityResolver: Interfaz que permite resolver los identificadores del sistema e identificadores públicos asociados con entidades a los sitios reales donde dichas entidades están almacenadas.
- XML Reader: Representa la funcionalidad básica que deben seguir los parsers que leen documentos y generan eventos SAX.
- XMLFilter: Interfaz que extiende a XMLReader para facilitar la composición de procesadores SAX. Las implementaciones de XMLFilter permiten construir un nuevo procesador SAX a partir de otro procesador SAX o procesador padre.

La clase DefaultHandler proporciona una implementación estándar para las cuatro primeras interfaces.

2.3 Analizadores GLR (G*eneralized* L*eft* to R*ight* Parsers)

2.3.1 Introducción

El mecanismo de análisis sintáctico es un componente fundamental en el procesamiento de lenguajes. Podemos clasificar los algoritmos de análisis sintáctico en dos grandes grupos, según el ámbito de gramáticas que pueden manejar:

- Algoritmos *específicos*, enfocados al manejo de clases específicas de gramáticas incontextuales, como las LL(k) o las LR(k) [Aho et al. 2007]. Los algoritmos de este grupo son muy eficientes por ser altamente deterministas y no necesitar, normalmente, realizar ningún tipo de búsqueda o vuelta-atrás durante el procesamiento. Este tipo de algoritmos se utilizan durante el procesamiento de muchos lenguajes de programación, al poder describirse sus sintaxis con gramáticas pertenecientes a alguna de las clases citadas.
- Algoritmos *generales*, enfocados al manejo de cualquier clase de gramática incontextual, como el algoritmo de Earley [Earley 1968] o el de Cocke, Younger y

Kasami [Younger 1967]. Dado que estos algoritmos deben tratar con fenómenos que, como el de la ambigüedad, no suceden en las gramáticas correspondientes a las clases referidas anteriormente, su complejidad es normalmente mayor.

El algoritmo GLR (G*eneralized* L*eft* to R*ight*) propuesto por Masaru Tomita [Tomita 1986] es uno de los algoritmos generales más utilizados, debido a las ventajas que presenta en comparación a los demás algoritmos propuestos para el mismo fin. Aunque, al contrario que otros algoritmos generales, el algoritmo no es capaz de tratar *cualquier* gramática, sí que permite tratar muchas de las ambigüedades que presentan las gramáticas usuales de una manera eficiente. Además, se basa en el algoritmo de análisis sintáctico LR convencional y su proceso es guiado por las mismas tablas de análisis generadas para un analizador sintáctico LR [Aho et al. 2007]. Las acciones conflictivas o no deterministas que, en forma de dos o más acciones en una misma celda, se presentan en las tablas, se tratan mediante la sustitución de la pila de análisis LR original por múltiples pilas combinadas en una estructura de grafo, denominada G*raph*-S*tructured* S*tack* (GSS). Además, los diversos árboles de análisis sintáctico para una misma sentencia que pueden derivarse al manejar gramáticas ambiguas se consiguen representar de manera completa y reducida en espacio mediante una estructura de grafo denominada S*hared*-P*acked* P*arse* F*orest* (SPPF).

En este proyecto de máster se adoptará el algoritmo GLR como motor básico de procesamiento, a fin de soportar especificaciones modulares de tareas de procesamiento de documentos XML. Efectivamente, la descomposición de una tarea en subtareas puede dar lugar a la realización de múltiples tareas sobre el mismo fragmento del documento. Las estructuras sintácticas demandadas por cada una de estas subtareas pueden, a su vez, diferir entre sí. Esto conduce a gramáticas ambiguas, lo que implica el uso de un algoritmo de análisis sintáctico general para soportar su manejo. Dado que nuestro trabajo previo [Sarasa et al. 2009d] ha utilizado métodos de análisis ascendentes, basados en gramáticas LALR(1), la elección del algoritmo GLR, que generaliza de manera natural estos métodos, se considera una opción lógica sobre la que soportar las propuestas realizadas en este proyecto. En esta sección se desarrolla un estudio a fondo del funcionamiento de este algoritmo. Dicho estudio se basa, además, en una herramienta que se ha desarrollado también en el contexto del proyecto, y que permite estudiar el comportamiento paso a paso del algoritmo. De la misma manera, además del estudio realizado para identificar las diferentes peculiaridades que presenta el algoritmo, se realiza un estudio de las mejoras introducidas en el mismo a lo largo de los años.

2.3.2 Visualización del comportamiento del algoritmo GLR

Durante los últimos años se han propuesto diferentes mejoras o extensiones del algoritmo de Tomita, así como ligeras modificaciones a la hora de adaptarlo a usos específicos. A fin de realizar un estudio detallado del funcionamiento del algoritmo, en este proyecto se ha desarrollado una herramienta de visualización que muestra cada uno de los estados que va adoptando la GSS y el SPPF a medida que se realiza el procesamiento (Figura 2.3.1). La herramienta muestra paso a paso el funcionamiento del algoritmo de Tomita para una

gramática y una sentencia de entrada. El visualizador procesa la traza que genera la herramienta GTB (G*rammar* T*ool* B*ox*) [Johnstone et al. 2004]. GTB ofrece un conjunto de funcionalidades para el manejo de gramáticas incontextuales. GTB posee una implementación mejorada (correcta, véase la sección 2.3.10) del algoritmo GLR. La gramática y cadena a analizar se escriben de acuerdo a la notación GTB, y se procesan con esta herramienta. Esto genera una traza del proceso junto al estado final de la GSS. Aunque el GTB no proporciona las tablas de análisis, ni el autómata característico de la gramática (véase la sección 2.3.3), sobre sus trazas se puede extraer y deducir la información necesaria para reconstruir cada paso del proceso.

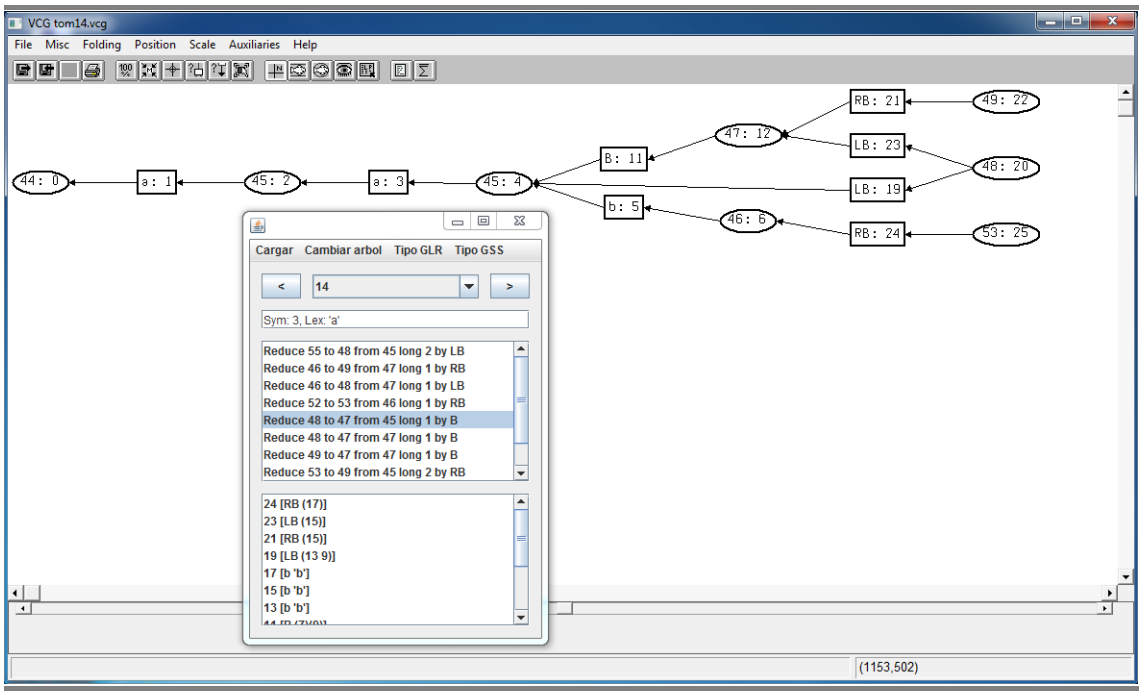


Figura 2.3.1. El Visualizador.

El visualizador es capaz de reconstruir dicha información permitiendo mostrar el estado de la GSS y del SPPF en un momento concreto de ejecución. Los grafos se imprimen en formato de texto, siendo capaces de ser configurados e interpretados por diferentes herramientas de visualización de grafos. La herramienta de visualización utilizada ha sido VCG (V*isualization* of C*ompiler* G*raphs*) [Lemke 1993].

La representación de los grafos y la traza del SPPF están representadas de la misma manera planteada por Tomita en [Tomita 1986] para explicar, de manera visual, los principales conceptos en los que se basaba su algoritmo. Las figuras y trazas mostradas en esta sección se han obtenido con ayuda del visualizador (por motivos de mejora de la presentación, los gráficos no son los directamente producidos por el visualizador, sino que se han redibujado; no obstante, sin la ayuda de la herramienta, la obtención de los mismos hubiera mucho más laboriosa y mucho menos fiable).

2.3.3 Análisis ascendente

Los algoritmos de análisis ascendente son aquellos que recrean el árbol de análisis sintáctico correspondiente a una sentencia partiendo de las hojas y progresando hacia el nodo raíz del árbol. Los algoritmos de tipo ascendente más comúnmente utilizados se caracterizan por ser de tipo *desplazamiento-reducción*. Este tipo de algoritmos reconstruyen, en sentido inverso, la *derivación más a la derecha* de la sentencia de entrada [Aho et al. 2007]. Para ello, supóngase que, al analizar una sentencia *w*, el algoritmo ha reconstruido ya un fragmento de derivación de la forma $\alpha_1 \equiv \beta\gamma y \Rightarrow \alpha_2 \Rightarrow \ldots \Rightarrow \alpha_n \equiv w$, y que el anterior paso de derivación es de la forma $\beta A y \Rightarrow \alpha_1$ (por lo que $A ::= \gamma$ ha de ser una producción). Entonces, el algoritmo mantiene la siguiente información de configuración:

- Una *pila de análisis*, en la que se encuentra apilada una forma sentencial φ, que es un prefijo de $\beta\gamma$ (un *prefijo viable*).
- El resto de la entrada que queda por analizar.

Entonces, el analizador puede realizar alguno de los dos siguientes tipos de movimientos:

- Un *desplazamiento*. Este movimiento ocurre cuando el analizador determina que *es posible* que en la pila de análisis no aparezca aún $\beta\gamma$. El movimiento consiste en el apilado del primer símbolo terminal en la entrada, en la pila de análisis sintáctico.
- Una *reducción*. Este movimiento ocurre cuando en la pila de análisis el analizador determina que en la pila de análisis *es posible* que aparezca ya $\beta\gamma$. El movimiento consiste en sustituir la subcadena γ que encaja con el cuerpo de la producción $A ::= \gamma$ que interviene en el paso de derivación previo (esta cadena se denomina *asidero*), por el símbolo no terminal A de la cabeza de dicha producción.

Para detectar la terminación del proceso, es posible añadir una nueva regla a la gramática, de la forma $S' \rightarrow S\$$, donde S es el axioma de la gramática original, S' el nuevo axioma, y $ un nuevo terminal denominado *símbolo de terminación* (una marca de fin de cadena). El análisis termina cuando se reduce dicha regla. Así mismo, en aquellas situaciones en las que el analizador no pueda llevar a cabo ningún movimiento, se anunciará un error, decidiéndose que la sentencia no es generada por la gramática.

Para determinar qué movimientos realizar, el analizador utiliza toda la información del prefijo viable φ que aparece en la pila, así como los *k* siguientes símbolos en la entrada (*símbolos de preanálisis*). De esta forma, habrá situaciones *conflictivas* en las que el analizador contemple simultáneamente la posibilidad de realizar un desplazamiento, y a la vez una reducción (conflicto de *desplazamiento – reducción*), así como situaciones en las que el analizador contemple la posibilidad de realizar dos reducciones (conflicto de *reducción – reducción*). En estos casos, el algoritmo debe, bien explorar todas las posibilidades, bien resolver los conflictos arbitrariamente o apoyándose en información extragramatical (p.ej.,

información sobre precedencias y asociatividades de operadores en expresiones). Existe una amplia clase de gramáticas (gramáticas LR(*k*) [Knuth 1965]) para las que es posible evitar este tipo de conflictos. No obstante, en el caso de tratar con gramáticas ambiguas, los conflictos son inevitables, ya que habrá cadenas que admiten dos o más derivaciones más a la derecha, ambas igualmente válidas.

Los algoritmos LR son un ejemplo importante de algoritmos de análisis ascendente por desplazamiento-reducción que funcionan con gramáticas LR(*k*) (siendo, normalmente, *k* = 1). Estos algoritmos se fundamentan en el hecho básico de que el conjunto de los prefijos viables que pueden aparecer en la pila de análisis sintáctico de un analizador por desplazamiento – reducción forman un *conjunto regular* [Knuth 1965]. Por tanto, es posible construir un autómata finito determinista que reconozca dichos prefijos viables: el *autómata característico* de la gramática. De esta forma, los algoritmos LR se guían por una *tabla de análisis* para realizar el procesamiento, que, en realidad, resulta de anotar con acciones la tabla de transición del autómata característico. Así mismo, estos algoritmos intercalan, en la pila de análisis sintáctico, símbolos de estado con símbolos gramaticales, con el fin de evitar tener que *ejecutar* el autómata característico sobre la pila para determinar el estado actual en el reconocimiento del prefijo viable contenido en la misma[1].

Existen múltiples métodos para construir dicho autómata. Dichos métodos introducen más o menos estados, y añaden información adicional a los estados a fin de determinar los posibles movimientos, diferenciándose en su potencia para evitar conflictos. De estos métodos, el método LALR (*Lookahead* LR) [DeRemer 1969] es el más ampliamente utilizado en la práctica.

Los estados de los autómatas característicos construidos por el método LALR (*autómatas* LALR(1)) se corresponden con conjuntos de *elementos* LALR(1): conceptualmente, pares de la forma $[A::=\alpha.\beta,\Phi]$, donde $A::=\alpha\beta$ es una producción, y Φ es un conjunto de terminales denominados *símbolos de preanálisis* del elemento. La idea es que, si, ante una configuración de la pila, el autómata se encuentra en un estado en el que hay un elemento de la forma $[A ::=\gamma.,\Phi]$, y el siguiente símbolo en la entrada es un terminal *a* que está en Φ, entonces es posible reducir por la regla $A ::=\gamma$. Esto no evita necesariamente los conflictos, ya que en ese mismo estado puede haber otro elemento de la forma $[B ::=\alpha.a\beta,\Omega]$ (lo que permitiría también *desplazar a* a la cima de la pila de análisis), o bien $[B::=\eta.,\Omega]$, con *a* en Ω, lo que permitiría también *reducir* por $B ::=\eta$. No obstante, para una amplia clase de gramáticas (las gramáticas LALR(1)), el método evita los conflictos (véase [Aho et al. 2007] para más detalle).

¿Cuáles son exactamente los estados de un autómata LALR(1)? Conceptualmente, la manera más fácil de *caracterizarlos* es [Grune & Jacobs 2008]:

- Partir de un autómata finito *no determinista* reconocedor de prefijos viables LR(1). Los estados de este autómata son *elementos* LR(1) de la forma $[A ::=\alpha.\beta,a]$, con *a* un terminal. Las transiciones son de dos tipos: $[A ::=\alpha.X\beta,a]=X=>[A ::=\alpha X.\beta,a]$, y [A

[1] Desde un punto de vista exclusivamente de reconocimiento, los símbolos gramaticales no serían necesarios. No obstante, desde un punto de vista de procesamiento, dichos símbolos pueden ser útiles para albergar información semántica adicional (p.ej., atributos semánticos).

::=α.Bβ,a]=λ=>[B::=.γ,b], para cada terminal b en los *primeros* de βa (un terminal *b* es un primero de una cadena α cuando de α se puede derivar una cadena que comienza por *b*; formalmente, α=>* *b*α').

- Aplicar a este autómata la *construcción por subconjuntos* [Aho et al. 2007] para obtener un autómata finito determinista equivalente: el autómata LR(1). En este autómata, cada estado estará etiquetado por conjuntos de elementos LR(1). Si tomamos, de estos elementos, las primeras componentes (es decir, los A $\rightarrow\alpha.\beta$), obtenemos lo que se llama *el corazón* del estado.

- El autómata LALR(1) se obtiene, entonces, *fusionando* los estados del autómata LR(1) que tienen el mismo corazón. Los símbolos de preanálisis de cada elemento LALR(1) integrarán los segundos componentes de los correspondientes elementos LR(1) que lo han originado.

No obstante, esta caracterización conceptual, aunque computable, es ineficiente. En la práctica se utilizan algoritmos más eficientes para construir estos autómatas (consúltese [Aho et al. 2007] para más detalle).

Disponiendo del autómata y conociendo el estado en el que nos encontramos, podemos conocer qué operación u operaciones son posibles aplicar en dicho estado. Dichas operaciones pueden codificarse en forma de una *tabla de análisis*:

- Si desde un estado parte una transición etiquetada por un terminal, dicha transición se corresponderá con un desplazamiento. En la tabla de análisis, se anota la correspondiente transición con una instrucción *shift* (desplazamiento), que viene acompañada de un número correspondiente al siguiente estado que se transita.

- Si un estado contiene un elemento de la forma [A ::= α.,Φ], las entradas en la tabla para cada símbolo terminal en Φ se anotarán con una acción *reduce* (reducción), que viene acompañada por una referencia a la regla A::= α. De esta forma, cuando el analizador descubra dicha acción, desapilará 2|α| elementos de la pila de análisis (esto se debe a que, como se ha indicado anteriormente, el analizador intercala estados con símbolos gramaticales en dicha pila), y realizará la transición correspondiente a A en el estado que se descubre tras realizar dichos desapilados.

- Si un estado contiene un elemento de la forma [S' ::= S, {$}], en lugar de almacenar en la entrada correspondiente una acción de reducción, se deberá almacenar una acción *accept* (aceptación de la entrada).

Obsérvese que, dada la tabla de transiciones del autómata característico, las acciones únicamente anotarán las entradas correspondientes a terminales (y al símbolo $). Por tanto, la tabla de análisis puede dividirse en dos subtablas: tabla *Action* y tabla *Goto*. Las filas de estas

tablas se corresponden con estados, y las columnas con símbolos gramaticales (terminales y $ en la tabla *Action*, y *no terminales* en la tabla *Goto*). Los valores de las celdas de *Action* reflejan las operaciones de desplazamiento, reducción, y acepción (aquellas celdas vacías se corresponderán con la detección de un *error* en el proceso de análisis). Los valores de *Goto* indicarán el siguiente estado al que se transita ante la visión de los no terminales que resultan al realizar las acciones de reducción. En términos de las tablas *Action* y *Goto* es sencillo describir el comportamiento del algoritmo LR (véase la Figura 2.3.2). Dicho algoritmo asume que la tabla *Action* no tiene celdas con múltiples entradas (es decir, que la gramática de partida no presenta conflictos).

Nótese que, si la gramática no es LALR(1), en una misma entrada de la tabla aparecerán dos o más acciones, que se corresponderán con conflictos en el proceso de análisis. Aunque estas tablas no pueden ser manejadas por el algoritmo LR estándar, sí pueden ser manejadas por el algoritmo GLR.

```
pila := {estadoInicial};
i := 0;
Repetir siempre
   estado := desapila(pila);
   Si ACTION[estado, input_i] = shift s entonces
      apila(input_i, pila);  apila(s, pila);
       i := i+1;
   sino si ACTION[estado, input_i] = reduce P entonces
      desapilaVarios(pila,2*|P.cuerpo|);
      s := cima(pila); apila(P.cabeza, pila); apila(GOTO[s, A], pila);
   sino si ACTION[estado, input_i] = accept entonces -éxito-
   sino -error-
```

Figura 2.3.2. Algoritmo LR.

0. S’ ::= S
1. S ::= 'a' S
2. S ::= B S
3. S ::= λ
4. B ::= LB
5. B ::= RB
6. LB ::= LB 'b'
7. LB ::= 'b'
8. RB ::= 'b' RB
9. RB ::= 'b'

Figura 2.3.3. Gramática de ejemplo.

A fin de ejemplificar la construcción realizada por el método LALR, se considerará la gramática ejemplo que se muestra en la Figura 2.3.3. La Figura 2.3.4 muestra el autómata LALR(1) construido por el método LALR para esta gramática.

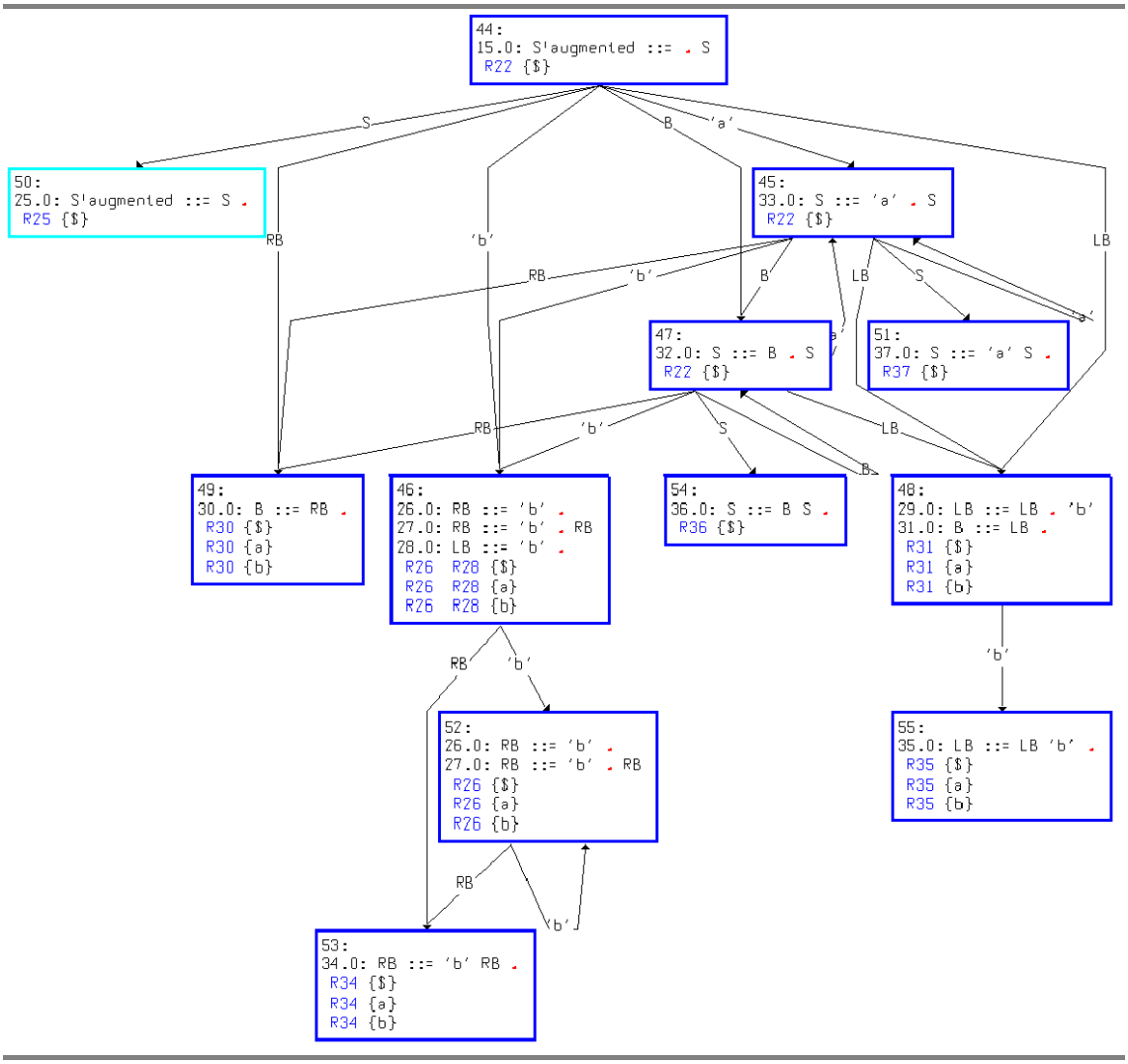


Figura 2.3.4. Autómata LALR(1) correspondiente a la gramática de ejemplo (obtenido mediante la herramienta GTB).

La tabla de análisis correspondiente a la gramática de ejemplo se muestra en la Figura 2.3.5. Esta tabla se ha generado a partir de la construcción del autómata de la Figura 2.3.5. Como puede observarse, dicha tabla contiene celdas con múltiples acciones. Para manejarla será necesario utilizar un algoritmo general, como el algoritmo GLR, que se examinará a continuación.

State	Action			Goto			
	'a'	'b'	$	S	B	LB	RB
44	Sh45	Sh46	Re3	50	47	48	49
45	Sh45	Sh46	Re3	51	47	48	49
46	Re9/Re7	Re9/Re7/Sh52					53
47	Sh45	Sh46	Re3	54	47	48	49
48	Re4	Re4/Sh55					
49	Re5	Re5					
50			Acc				
51			Re1				
52	Re9	Re9/Sh52					
53	Re8	Re8					
54			Re2				
55	Re6	Re6					

Figura 2.3.5. Tabla de análisis correspondiente a la gramática de ejemplo (obtenida mediante la herramienta GTB; se ha respetado la numeración de estados realizada por dicha herramienta).

2.3.4 El algoritmo GLR

El algoritmo de Tomita parte de la idea de procesamiento utilizada en el algoritmo LR. Tal y como se ha discutido en la sección anterior, dependiendo del método de generación de tablas utilizado, las tablas de análisis de un analizador LR pueden presentar *conflictos*, que se reflejarán como múltiples acciones para una misma entrada de la tabla. Como hemos visto, la tabla de análisis de la sección 2.3.3 presenta conflictos por haber sido construida por el método LALR a partir de una gramática no LALR(1). El algoritmo LR no puede manejar dichas entradas con múltiples acciones. Sin embargo, el algoritmo GLR sí es capaz de guiarse mediante estas tablas con conflictos. La idea básica es que, ante un conflicto, el algoritmo *bifurca* la pila de análisis. A fin de evitar una proliferación exponencial de pilas, el algoritmo combina todas ellas en una estructura en forma de grafo: la ya citada G*raph* S*tructured* S*tack* (GSS). Dado que el algoritmo puede tratar gramáticas ambiguas, para la representación de los diversos árboles que pueden generarse durante el análisis de las sentencias derivadas a partir de tales gramáticas, se utiliza otra estructura de grafo: el ya citado S*hared*-P*acked* P*arse* F*orest* (SPPF).

2.3.5 La Graph Structured Stack (GSS)

El algoritmo LR convencional es completamente determinista, y utiliza una única pila durante el análisis de las sentencias. El tratamiento de tablas con conflictos requiere, sin embargo, la existencia de varias pilas. El algoritmo GLR combina múltiples pilas en una estructura de grafo que le permite realizar un procesamiento eficiente con reducido coste en espacio. Esta estructura se denomina G*raph* S*tructured* S*tack* (GSS).

La GSS es un grafo acíclico dirigido. Se compone de vértices *estado* (en la representación visual se dibujarán con círculos), y vértices *símbolo* (representados por cuadrados en la representación visual). Un vértice *estado* representa un estado del autómata característico de

la gramática (dicho autómata puede haber sido construido por cualesquiera de los métodos aludidos en la sección anterior, y, en particular, por el método LALR). Un vértice *símbolo* representa un símbolo de la gramática, terminal o no-terminal. Contiene un puntero al nodo correspondiente en el SPPF. La manera en la que evoluciona este grafo se describe con profundidad en la sección 2.3.7.

Es importante señalar que, en la implementación del algoritmo, se preserva la GSS completa, aún cuando se realicen acciones de reducción (Figura 2.3.6). Sin embargo, tanto en la descripción conceptual del proceso realizada por Tomita, como en el visualizador que hemos implementado, se procede como si la GSS se podase como resultado de realizar reducciones. Puesto que mantener la GSS completa no es necesario para el correcto funcionamiento del algoritmo, dicha poda aumenta la eficiencia en espacio del algoritmo. Aunque en la propuesta realizada en este proyecto no se llevará a cabo, consideramos que esta optimización es crítica en nuestro contexto de aplicación, y que merece un análisis detallado en el trabajo a realizar en el futuro. Esto es debido a que el procesamiento de un documento XML que contenga una secuencia de cientos de miles de elementos puede generar un GSS con cientos de miles de nodos registrados en memoria. Esto puede provocar un coste en memoria proporcional a la anchura de dicho documento, lo que supone no sólo una pérdida importante en eficiencia, sino que incluso es capaz de impedir la terminación del proceso por agotamiento de recursos en documentos extremadamente anchos.

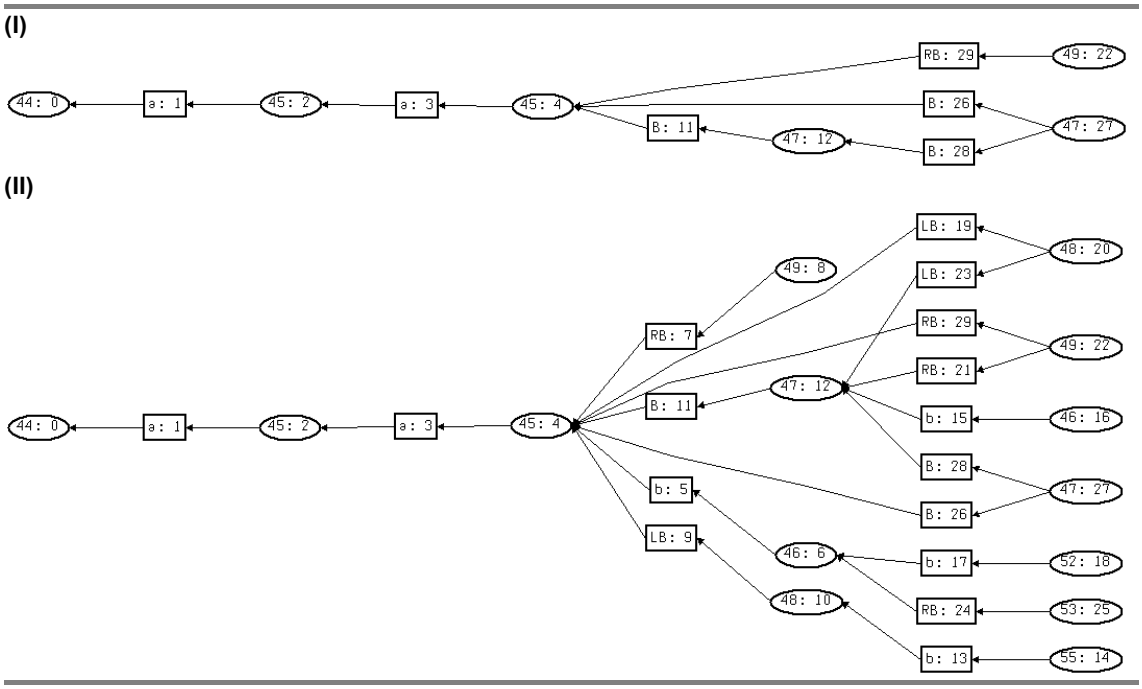


Figura 2.3.6. (I) Estado de la GSS visualizando sólo los nodos accesibles.
(II) Estado real de la GSS en la implementación canónica del algoritmo.

2.3.6 El Shared-Packed Parse Forest (SPPF)

El análisis de una sentencia con respecto a una gramática ambigua puede producir como resultado más de un árbol de análisis sintáctico. A fin de generar todos los árboles posibles y de ser capaz de almacenarlos en una estructura eficiente en espacio, el algoritmo GLR construye una estructura arborescente denominada S*hared*-P*acked* P*arse* F*orest* (SPPF). El SPPF es una representación eficiente de un bosque de árboles sintácticos que refleja, en una sola estructura, las distintas interpretaciones con las que se puede procesar una sentencia *ambigua* (una sentencia con dos o más árboles de análisis sintáctico asociados). Para ello, y a efectos de construir el SPPF, el algoritmo GLR no introduce los símbolos gramaticales correspondientes en la pila, como haría normalmente un algoritmo tipo LR convencional, sino que introduce punteros a nodos pertenecientes al SPPF. De esta forma:

- Con cada acción de desplazamiento no se introduce en la pila el símbolo terminal reconocido, sino que se crea una hoja en el SPPF con el lexema de la palabra y símbolo terminal correspondiente, y se introduce en la pila un puntero a dicha hoja.

- Por su parte, cada acción de reducción supone desapilar uno o varios elementos obteniendo punteros del SPPF. En este caso se crea un nuevo nodo en el SPPF con el símbolo no terminal en la parte izquierda de la regla utilizada en la reducción, o bien se utiliza uno apropiado previamente creado. Como enlaces, y como veremos más adelante, un nodo del SPPF dispone de un conjunto de listas de hijos. Los hijos de este nodo quedan determinados por los nodos a los que apuntaban los elementos desapilados (estos nodos representarán los subárboles asociados a los símbolos de la parte derecha de la regla), los cuales se añaden al conjunto como un nuevo enlace. Lo que se apila al finalizar la acción de reducción es el puntero al nodo aludido anteriormente.

El SPPF sigue, así mismo, dos criterios de optimización espacial: *compartición de sub-árboles* y *compactación por ambigüedad local*. De esta manera:

- La *compartición de sub-árboles* se basa en la idea de no repetir la representación de sub-árboles idénticos que aparezcan presentes en diferentes árboles sintácticos. No se crean nodos repetidos. Si existe un nodo en el SPPF con el mismo símbolo asociado, nivel en el árbol, y los mismos nodos hijos, se apila el puntero de dicho nodo sin crear una nueva copia.

- La *compactación por ambigüedad local* se fundamenta en el hecho de que un fragmento de la cadena de entrada presenta ambigüedad local si puede ser reducido a un mismo símbolo no terminal de dos o más formas distintas. Este problema implica un coste es espacio que puede llegar a ser exponencial para segmentos que presentan gran ambigüedad local. Para evitarlo, los nodos del árbol que representan el mismo símbolo no-terminal en un mismo nivel y que producen las mismas hojas, pueden *empaquetarse* como un único nodo. Esto

reduce el número de nodos y aristas que se utilizan a la hora de representar el resto de nodos padre. Los subárboles que se obtienen tomando como raíz un nodo empaquetado se mantienen idénticos. Obsérvese también que la estructura para representar estos nodos empaquetados deja de ser representable mediante una estructura de árbol tradicional. De esta forma, el SPPF se implementa como un grafo acíclico dirigido y ordenado, en donde cada vértice puede contener cero o varias listas de nodos sucesores.

El comportamiento de ambas optimizaciones queda reflejado en la Figura 2.3.7.

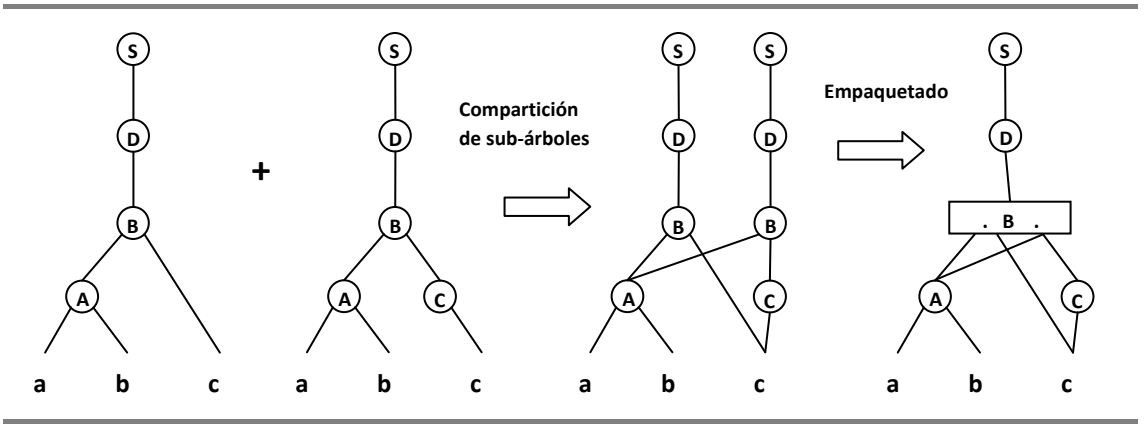


Figura 2.3.7. Compartición de sub-árboles y compactación por ambigüedad local.

2.3.7 Implementación del algoritmo

El algoritmo que se presenta a continuación es la versión final de las cuatro propuestas en [Tomita 1986], descrito de manera más comprensible. Este algoritmo se caracteriza por construir una GSS no podada al mismo tiempo que construye el SPPF. Principalmente, el algoritmo se compone de tres partes bien diferenciadas: una primera parte que determina y clasifica las posibles acciones ejecutables sobre los vértices de tipo *estado* a procesar (acciones de reducción o desplazamiento), una segunda parte que ejecuta todas las acciones de reducción, y una última parte que ejecuta todas las acciones de desplazamiento.

La Figura 2.3.8 y Figura 2.3.9 muestra el algoritmo de Tomita en pseudocódigo. La notación utilizada para la creación de objetos se realiza mediante instrucciones *new*, con una semántica análoga a la de lenguajes tipo Java, devolviendo referencias a los objetos creados y no objetos en sí. En el algoritmo, la GSS se representa utilizando dos tipos de nodos:

- Objetos de tipo *NodoSimbolo:* representan los nodos de tipo símbolo. Poseen como atributos: *nodoSPPF* (referencia al nodo en el SPPF), *prev* (conjunto de nodos predecesores al actual en la GSS), y *sig* (conjunto de nodos siguientes en la GSS).

- Objetos de tipo *NodoEstado:* representan los nodos de tipo estado. Poseen como atributos: *estado* (identificador del estado), *prev* (conjunto de nodos predecesores al actual en la GSS), y *sig* (conjunto de nodos siguientes en la GSS).

El SPPF, por su parte, se representa utilizando los siguientes tipos de nodos:

- Objetos de tipo *nodoNT:* representan nodos de tipo *no terminal*. Poseen como atributos: *nt* (símbolo no terminal), e *hijos* (conjunto de listas de hijos). La cadena vacía (λ) se representa mediante una lista de hijos vacía.

- Objetos de tipo *NodoT:* representan nodos de tipo *terminal*. El símbolo terminal es el valor del atributo *t*.

Otras estructuras importantes son:

- Lista *Estados:* contiene los objetos de tipo *estados* en la GSS en los que se encuentra el proceso de análisis.

- Conjunto *Reducciones:* almacena objetos de tipo *AccionReduccion*. Estos objetos poseen como atributos: *origen* (referencia al nodo de tipo estado en la GSS), *símbolo* (cabeza de la producción utilizada en la reducción), y *longitud* (número de símbolos del cuerpo de dicha producción). Esta información es necesaria para la ejecución de las reducciones (líneas 24-68 de la Figura 2.3.8).

- Pila C*ontextosReducciones* y pila *ContextosAplicables*: almacenan objetos de tipo *ContextoReducción.* Estos objetos poseen como atributos: *i* (número de símbolos que quedan por desapilar), *estado* (referencia al estado actual en la GSS), e *hijos* (lista de referencias a nodos en el SPPF).

- Conjunto *Desplazamientos:* almacena objetos de tipo *AccionDesplazamiento.* Estos objetos poseen como atributos: *origen* (referencia al nodo de tipo estado en la GSS), y *destino* (identificador de estado al que se transita como consecuencia del desplazamiento). Esta información es necesaria para la ejecución de los desplazamientos (líneas 69-83 de la Figura 2.3.9).

Para el acceso a la información de la tabla de análisis se utilizan las notaciones siguientes:

- Para el acceso a la tabla Action se utiliza la notación *Accion[identificador estado, símbolo terminal]*.

- Para la el acceso a la tabla Goto se utiliza *GOTO[identificador estado, símbolo no terminal]*.

Las producciones de la gramática se acceden mediante la notación *Producción*, siendo posible recuperar el símbolo cabeza de la producción mediante su atributo *cabeza* y los símbolos del cuerpo mediante el atributo *cuerpo*.

```
1.  Estados := [new Estado(estado: S0, prev: ∅, sig: ∅)];
2.  Aceptada := false;
3.  i := 0
4.  Mientras Estados ≠ [] hacer
5.    Repetir
6.      Reducciones := []
7.      Desplazamientos := []
8.      Mientras Estados ≠ []
9.        n := first Estados; Estados := rest Estados;
10.       Para cada a en Accion[n.estado, inputi]
11.         Si a.tipo = shift entonces
12.           Desplazamientos := Desplazamientos ++ new AccionDesplazamiento(origen: n, destino: a.estado)
13.         si no, si a.tipo = reduce entonces
14.           Reducciones := Reducciones ++ new AccionReduccion(origen: n, símbolo:
15.                          Producciones[a.produccion].cabeza, longitud: |Producciones[a.produccion].cuerpo|)
16.         si a.tipo = accept entonces
17.           Aceptada := true;
18.       fin Para
19.     fin Mientras
20.
21.     Para cada r en Reducciones
22.       ContextosReducciones:= [new ContextoReduccion(i: r.longitud, estado: r.origen, hijos:[])]
23.       ContextoAplicables  := []
24.       Mientras ContextosReducciones  ≠ []
25.         ContextoActual := first ContextosReducciones; ContextosReducciones := rest ContextosReducciones;
26.         Si ContextoActual.i = 0
27.           Si GOTO[r.origen.estado, r.símbolo] ≠ ⊥ /* contiene una entrada */
28.             ContextosAplicables := ContextosAplicables ++ ContextoActual
29.         si no
30.           Para cada Simbolo en ContextoActual.r.prev
31.             Para cada Estado en Simbolo.prev
32.               ContextosReducciones := ContextosReducciones ++
33.                                       new ContextoReduccion(i:ContextoActual-1, estado:Estado,
34.                                                             hijos:ContextoActual.hijos++Simbolo.nodo)
35.       Fin Mientras
36.
37.       Para cada c en ContextosAplicables
38.         EstadoDestino := n |  ∃ n'∈ c.estado.sig, ∃n''∈ n'.sig: n = n''.sig y n.estado =  GOTO[c.estado, r.símbolo]
39.         Si EstadoDestino ≠ ⊥
40.           Simbolo := n | n∈EstadoDestino.prev y n.nodo.simbolo = r.simbolo
41.           Si Simbolo ≠ ⊥
42.             Simbolo.prev := Simbolo.prev ∪ {c.estado}
43.             Simbolo.nodoSPPF.hijos := Simbolo.nodoSPPF.hijos ∪ {c.hijos}
44.             c.estado.sig := Contexto.estado.sig ∪ {Simbolo}
45.           si no,
46.             Simbolo := new Simbolo(nodo: new nodoNT(nt: r.simbolo, hijos: {r.hijos}), prev: {c.estado},
47.                                    sig: {EstadoDestino})
48.             EstadoDestino.prev := EstadoDestino.prev ∪ {Simbolo}
49.             c.estado.sig := Contexto.estado.sig ∪ {Simbolo}
50.         si no,
51.           EstadoDestino := new Estado(estado: GOTO[c.estado, r.símbolo], sig: ∅)
52.           Simbolo := new Simbolo(nodo: new nodoNT(nt: r.simbolo, hijos: {r.hijos}), prev: {c.estado},
53.                                  sig: {EstadoDestino})
54.           EstadoDestino.prev := {Simbolo}
55.           c.estado.sig := c.estado.sig ∪ {Simbolo}
56.           Estados := Estados ∪ {EstadoDestino}
57.       fin Para
58.     fin Para
59.   hasta que Estados = ∅
60.
61.
62.
63.
64.
65.
66.
67.
68.
```

Figura 2.3.8. Algoritmo GLR (parte I).

```
69.     t := new NodoT(t:input_i);
70.     Para cada d en Desplazamientos
71.       SiguienteNodoEstado := n | n en Estados tal que n.estado = d.destino
72.       Si SiguienteNodoEstado ≠ ⊥
73.           NodoSimbolo := n | n en SiguienteNodoEstado.prev;
74.           NodoSimbolo.prev := NodoSimbolo.prev ∪ {d.origen};
75.           d.origen.sig := {NodoSimbolo}
76.       si no,
77.           NodoSimbolo := new NodoSimbolo(nodo: t, prev: {d.origen})
78.           NodoEstado := new NodoEstado(estado: d.destino, prev: {NodoSimbolo}, sig: ∅)
79.           NodoSimbolo.sig := {NodoEstado}
80.           Estados := Estados ∪ {NodoEstado}
81.     fin para
82.     i := i+1
83.   fin Mientras
84.
85.
```

Figura 2.3.9. Algoritmo GLR (parte II).

Inicialmente, *Estados* se fija a una referencia al estado inicial del autómata característico (línea 1). El proceso continúa mientras existan estados de análisis susceptibles de ser explorados (líneas 4-85).

Cada iteración del algoritmo consta de tres fases consecutivas:

- Determinación de todas las posibles acciones (desplazamientos y reducciones) a realizar sobre los estados de *Estados* (líneas 9-22). Para cada estado, y en base al símbolo de entrada actual, se consulta la correspondiente entrada de la tabla *Acción*, y se almacena en los conjuntos *Reducciones* y *Desplazamientos* la información necesaria para ejecutar, posteriormente, dichas acciones. Para ello se utilizan objetos de tipos, respectivamente, *AccionReduccion* y *AccionDesplazamiento*. Así mismo, si alguna de las acciones es de aceptación (*accept*), el algoritmo registra dicho hecho a fin de dejar constancia de que la entrada se ha reconocido como parte del lenguaje generado por la gramática (línea 20).

- Ejecución de todas las reducciones posibles (líneas 24-68). El caso de reducciones nulas también es contemplado en esta fase. Esta fase se divide en dos subfases:

  - Realizar el *desapilado* requerido por cada acción de reducción. Para cada acción se determina el conjunto posible de estados al que se llega tras desapilar el número de símbolos que indica la acción, y, para cada uno de estos estados, la lista de referencias a nodos en el SPPF contenidas en los símbolos desapilados. Esta información se determina en las líneas 24-40 mediante un algoritmo dirigido por una pila de *contextos de reducción (ContextosReducciones).* Por cada reducción, se determina un *ContextoReduccion* y se apila en *ContextosReducciones*. En cada iteración, se desapila un contexto de reducción. Si el contador de símbolos por

desapilar en este contexto es 0, se comprueba si, desde el estado del contexto, es posible realizar la transición que termina la reducción. Si es así, se guarda dicho contexto (en *ContextosAplicables*) para realizar, posteriormente, la reducción. Si el contador no es 0, se obtienen los estados que preceden al actual, y, para cada uno de ellos, se crea un nuevo contexto de reducción (líneas 34-39).

- Realizar las transiciones. Para cada contexto, se determina primeramente si el estado al que debe transitarse ya ha sido creado. Si es así, si el no terminal que produce la transición también ha sido ya insertado, se inserta en el nodo SPPF asociado al mismo la lista de hijos, y se vincula dicho símbolo con el estado de origen (líneas 45-51). En caso de que el símbolo no haya sido insertado, se crea dicho símbolo, el correspondiente no terminal en el SPPF, y se vincula tanto con el estado de partida como con el estado al que se transita (líneas 52-57). En caso de que el estado no se haya creado, se crea tanto el estado, como el símbolo, como el no terminal en el SPPF, se vinculan entre sí todos estos objetos, y se almacena el estado en la lista de estados actuales (líneas 58-65).

El proceso iterativo finaliza cuando ya no hay más estados posibles, lo que supondrá que se han realizado todas las posibles reducciones.

- Ejecución de todos los desplazamientos posibles a los que da lugar el símbolo actual: líneas 69-83). Al desplazar desde cada estado, en caso de llegar a un estado ya contemplado, se reutilizan los correspondientes nodos en la GSS (líneas 72-76). En caso contrario, se crean nuevos nodos y se vinculan estos entre sí (línea 77-81), y el nuevo estado se almacena en la lista de estados posibles (línea 82). En cualquier caso, todos los nodos símbolo creados refieren el mismo nodo en el SPPF en esta fase. Por último, es importante resaltar que la GSS mantiene toda la información de los nodos, a diferencia de como se presentará su funcionamiento en la siguiente sección, en la que, por claridad de presentación, se eliminarán aquellos nodos que ya no son necesarios.

2.3.8 Ejemplo de Funcionamiento del algoritmo

El funcionamiento básico del algoritmo se describirá con un ejemplo realizado en el visualizador que hemos implementado sobre la herramienta GTB, aunque redibujado para facilitar la lectura. La gramática que utilizaremos es la gramática de ejemplo de la Figura 2.3.3. Procederemos a analizar la sentencia “aabba”. Como puede observarse, la gramática es ambigua. Existen varias formas de procesar la sub-sentencia “bb”: con recursión a izquierdas mediante LB, con recursión a derechas mediante RB o reduciendo el primer carácter ‘b’ por la regla LB o RB, y el siguiente carácter por la regla LB o RB. Los árboles sintácticos que se pueden derivar de esta gramática para la sentencia dada son exactamente seis. La Figura 2.3.10

muestra la evolución del algoritmo durante el análisis de dicha sentencia. De esta forma, el análisis de esta sentencia se inicia como sigue (en cada paso, se hace referencia a la correspondiente fila de la Figura 2.3.10).

- (A) Al comienzo del proceso, la GSS se compone únicamente del estado inicial (estado número 44 del autómata). El SPPF es, en este punto, vacío.

- (B) El siguiente movimiento reconoce el primer carácter 'a' de la sentencia de entrada. Esto produce un desplazamiento del estado 44 al estado 45. En el SPPF se crea un nodo con símbolo y contenido 'a'. El puntero a este nodo se denota por el número 1 (o $1), puntero que se almacena en el GSS (por claridad, el visualizador también indica el símbolo desplazado).

- (C) A continuación se desplaza el siguiente 'a'. Esto produce una transición del estado 45 al mismo estado 45, así como la creación de un nuevo nodo en el SPPF, que contiene el símbolo 'a'. El puntero a dicho vértice se denota por el número 2.

- (D) El siguiente movimiento es desplazar 'b'. Esto produce un desplazamiento del estado 45 al estado 46, así como la creación del correspondiente vértice en el SPPF (puntero número 3).

En este instante aparecen conflictos, causados en este caso por la ambigüedad de la gramática: un conflicto de reducción-reducción, así como dos conflictos de desplazamiento-reducción (véase la celda correspondiente al estado 46 y al terminal 'b' en la tabla de análisis de la Figura 2.3.5).

La presencia de conflictos en las tablas implica, en el caso más general, bifurcar la pila en el GSS. Estas bifurcaciones implican la pérdida del coste lineal del algoritmo. Para el manejo de estos casos, se ejecutan primero las acciones de reducción, tal y como se ha visto en la sección anterior. De esta manera, y tal y como se ha indicado también en la sección anterior, las acciones de desplazamiento sólo se podrán aplicar una vez realizadas todas y cada una de las acciones de reducción de todos y cada uno de los estados de la pila reducibles, hasta que ya no sea posible llevar a cabo más reducciones.

La elección de la acción de reducción a aplicar en cada caso se realiza de manera arbitraria. Las reducciones aplicables sobre un estado se calculan como una distancia fija que hay que recorrer o el número de vértices tipo *símbolo* que hay que desapilar del GSS a partir del estado sobre el que se práctica la reducción. Esta información puede obtenerse a partir de la producción referida por la correspondiente acción de reducción, como ya se discutió en la sección 2.3.3. Un aspecto importante es que, en cada nivel de la GSS, no pueden existir vértices estado repetidos. Sin embargo, si pueden existir vértices símbolo repetidos.

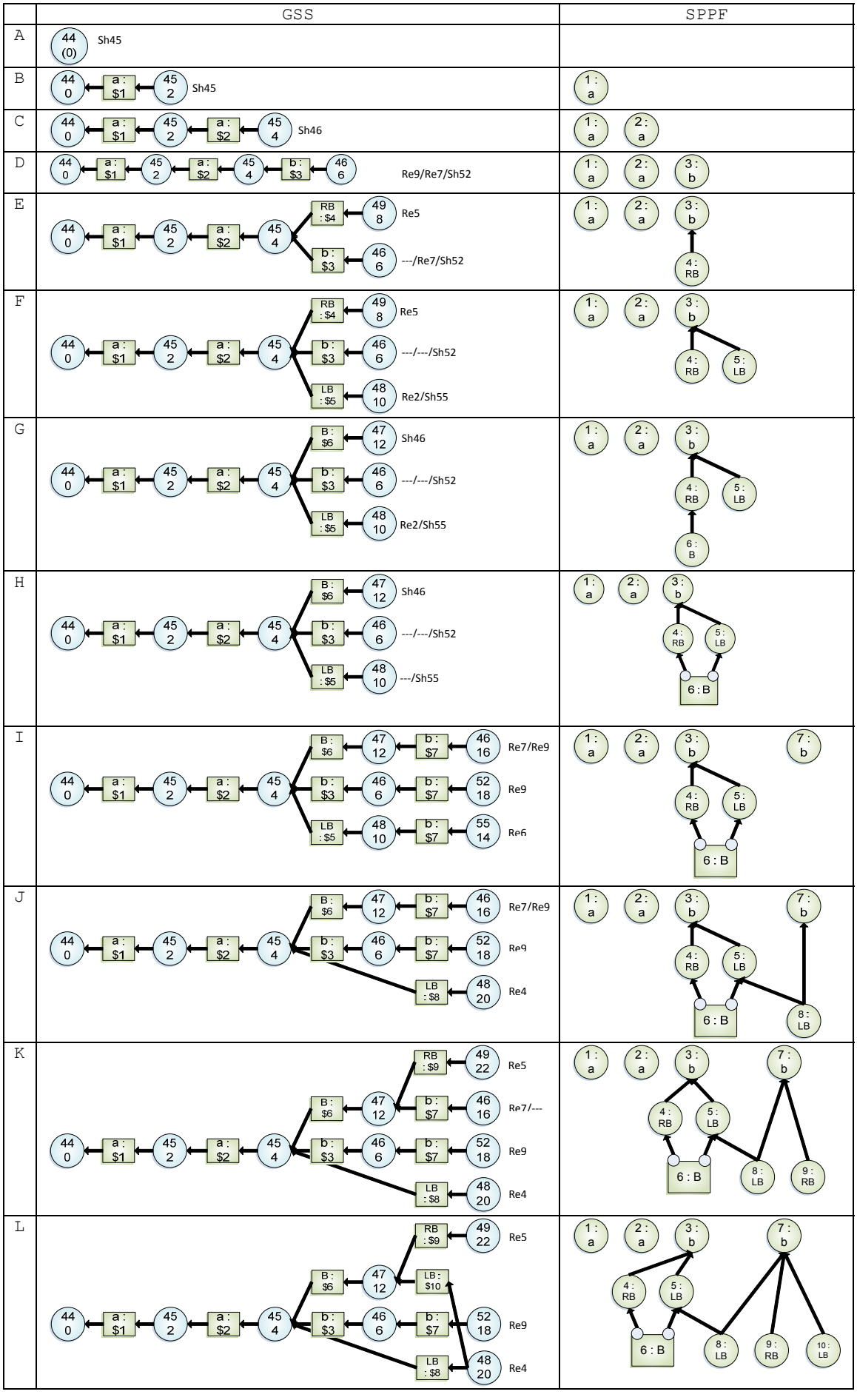


Figura 2.3.10. Evolución de la GSS y del SPPF.

De esta forma, en el ejemplo, el proceso continúa como sigue:

- (E) Se aplica la primera acción de reducción (Re9) sobre el estado 46 de longitud 1. Se llega al vértice estado número 4. Al reducir por la regla 9 (RB ::= 'b'), se genera el estado 49. El SPPF ahora contiene un nuevo nodo RB (su puntero se identifica por 4). Dicho nodo tendrá una única lista de hijos, que tendrá como único elemento el nodo apuntado por el vértice símbolo desapilado (nodo identificado por 3). Nótese que el vértice número 6 correspondiente al estado 46, junto a todos sus sucesores en la GSS, no pueden eliminarse todavía al existir una operación todavía por aplicar sobre dicho vértice.

- (F) Se aplica la segunda acción de reducción (Re7) sobre el estado 46 de longitud 1. Se llega al vértice estado número 4. Al reducir por la regla 7 (LB ::= 'b'), se genera el estado 48. El SPPF ahora contiene un nuevo nodo LB (su puntero se identifica por 5). Dicho nodo tendrá una única lista de hijos, que tendrá como único elemento el nodo apuntado por el vértice símbolo desapilado (nodo identificado por 3). El vértice estado número 6, junto a todos sus sucesores en la GSS, no pueden eliminarse (visualmente) todavía al existir aún una operación por aplicar sobre dicho vértice.

- (G) Se aplica la única acción de reducción (Re5) sobre el estado 49 de longitud 1. Se llega al vértice estado número 4. Al reducir por la regla 5 (B ::= RB), se genera el estado 47. El SPPF ahora contiene un nuevo nodo B (su puntero se identifica por 6). Dicho nodo tendrá una única lista de hijos, que tendrá como único elemento el nodo apuntado por el vértice símbolo desapilado (nodo identificado por 4). El vértice estado número 8, junto a todos sus sucesores en la GSS que no contengan un nodo padre distinto del vértice 8 en una longitud de 1, pueden eliminarse (visualmente) al haberse consumido todas las operaciones por aplicar sobre dicho vértice.

- (H) Se aplica la acción de reducción (Re4) sobre el estado 48 de longitud 1. Se llega al vértice estado número 4. Al reducir por la regla 4 (B ::= LB), se genera el estado 47. El estado 47 ya existe en la GSS. El SPPF ya contiene un nodo B, cuyo puntero es el 6. En este caso se produce una compactación por existencia de ambigüedad local. A su lista de hijos, se añade una nueva lista que tendrá como único elemento el nodo apuntado por el vértice símbolo desapilado (nodo identificado por 5). El vértice estado número 10, junto a todos sus sucesores en la GSS, no pueden eliminarse (visualmente) todavía al existir aún una acción de desplazamiento pendiente de aplicar sobre dicho vértice.

Es interesante, en este punto, analizar la forma en la que el algoritmo implementa los criterios de optimización espacial discutidos anteriormente:

- La compactación por ambigüedad local queda reflejada en el GSS como un proceso de fusión de vértices. Esta fusión se produce cuando el vértice estado

inmediatamente a la izquierda y el vértice estado inmediatamente a la derecha coinciden. El vértice símbolo será siempre el mismo, pero en el SPPF hay que reflejar el empaquetamiento debido a la existencia de una ambigüedad local. Esto implica actualizar la información de ese nodo, creando una nueva lista de hijos pertenecientes a los nodos desapilados. Esta clase de fusiones se denominarán *fusiones de tipo* I.

- Por otra parte, la compartición de sub-árboles se produce porque, cuando se genera un vértice símbolo con valor de etiqueta y lista de hijos iguales a los de un vértice símbolo ya existente en la GSS y se transita al mismo estado, se aprovecha el vértice símbolo existente en la GSS sin crear uno nuevo. Tampoco se crea ningún nodo nuevo en el SPPF. Sin embargo, esto sólo es posible cuando el vértice símbolo que se pretende generar, refiere al mismo nodo del SPPF, es decir, contiene el mismo valor de etiqueta y lista de nodos hijos. Esta clase de fusiones se denominarán *fusiones de tipo* II.

Nótese que es imprescindible en una fusión (ya sea ésta de tipo I, ya sea de tipo II) que el estado al que se transita sea siempre el mismo. El hecho de fusionar nodos en el GSS para reducir el coste en espacio no quiere decir que todas sus fusiones implican un empaquetamiento en el SPPF. Esa es la diferencia entre una fusión de tipo I y de tipo II. De hecho, como se vio en la sección 2.3.7, realmente no es necesario realizar ninguna fusión, sino que dichas representaciones se crean de manera directa durante el procesamiento. El proceso del ejemplo continúa como sigue:

- (I) El siguiente movimiento es desplazar ‘b’. Esto produce tres desplazamientos desde tres estados: un desplazamiento del estado 46 al estado 52, un desplazamiento del estado 48 al estado 55, y un desplazamiento del estado 47 al estado 46; así como la creación un único nodo que contiene ‘b’ en el SPPF, con tres vértices símbolo en la GSS que le apuntan (puntero identificado por 7). Aunque hemos avanzado tres pasos, y la GSS ha sufrido tres cambios, el SPPF solo ha experimenta un cambio.

- (J) El siguiente movimiento es reconocer el carácter ‘a’. Se aplica la única acción de reducción (Re6) sobre el estado 55 de longitud 2. Se llega al vértice estado número 4. Al reducir por la regla 9 (LB ::= LB ‘b’), se genera el estado 48. El SPPF ahora contiene un nuevo nodo LB (su puntero se identifica por 8). Dicho nodo tendrá una única lista de hijos, que tendrá como elementos los nodos apuntados por los vértices símbolo desapilados (nodo identificado por 5 y 7). El vértice estado número 14, junto a todos sus sucesores en la GSS que no contengan un nodo padre distinto del vértice 14 en una longitud de 2, pueden eliminarse (visualmente) al haberse consumido todas las operaciones por aplicar sobre dicho vértice.

- (K) Se aplica la segunda acción de reducción (Re9) sobre el estado 46 de longitud 1. Se llega al vértice estado número 12. Al reducir por la regla 9 (RB ::= ‘b’), se

genera el estado 49. El SPPF ahora contiene un nuevo nodo RB (su puntero se identifica por 9). Dicho nodo tendrá una única lista de hijos, que tendrá como único elemento el nodo apuntado por el vértice símbolo desapilado (nodo identificado por 7). El vértice número 16 correspondiente al estado 46, junto a todos sus sucesores en la GSS, no pueden eliminarse (visualmente) todavía al existir una operación todavía por aplicar sobre dicho vértice.

- (L) Se aplica la primera acción de reducción (Re7) sobre el estado 46 de longitud 1. Nótese que todas las acciones de reducción encoladas en un vértice estado poseen la misma longitud. Se llega al vértice estado número 12. Al reducir por la regla 7 (LB ::= 'b'), se genera el estado 48. El estado 48 ya existe en la GSS. El SPPF ya contiene el nodo LB, cuyo puntero es el 8. Aunque los vértices estado siempre se fusionan, en este caso se genera un vértice símbolo con un nuevo nodo en el SPPF (identificado con el puntero 10). No puede fusionarse por no cumplir la norma de fusión de tipo I. Tampoco lo cumple para el tipo II puesto que el contenido de este vértice símbolo difiere al existente. Están en contextos diferentes por provenir de vértices estados de etapas diferentes.

Avanzando en el proceso, ocurre un caso de reducción nula o reducción de longitud 0, tal y como se muestra en la Figura 2.3.11.

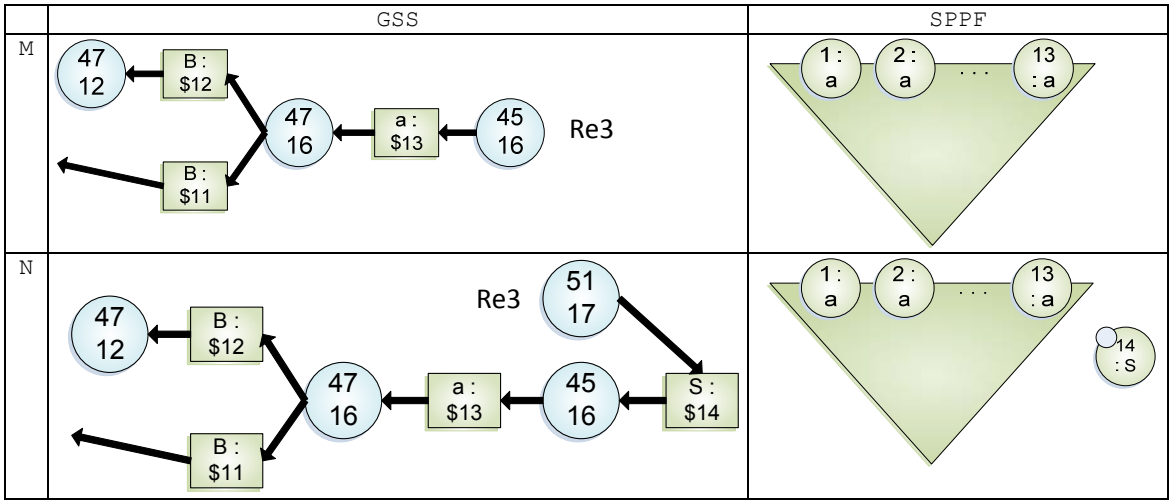


Figura 2.3.11. Comportamiento y estado de la GSS ante una reducción nula.

- (M) Se aplica la acción de reducción (Re3) sobre el estado 45 de longitud 0. Se llega al vértice estado número 16 (el mismo, es decir, no implica búsqueda de nodos sucesores). Al reducir por la regla 3 (S ::= λ), se genera el estado 51. Este vértice estado creado se corresponde con la etapa actual debido a que las etapas vienen determinadas por la palabra reconocida de la cadena de entrada y, las acciones a aplicar sobre dicho vértice, están todavía por aplicar durante esta etapa. En el SPPF se crea un nodo que contiene a S (su puntero se identifica por 14), cuya única lista de hijos es una lista vacía, que, de este modo representa la

reducción nula. (N) Muestra el estado de la GSS tras esta operación junto al estado del SPPF.

El resto del proceso que se sigue es análogo a los casos ya vistos. El proceso termina reconociendo con éxito la cadena de entrada cuando en la última etapa sólo quedan por aplicar las acciones de "acc" (aceptar). Pueden existir caminos que queden aislados, cortados y sin instrucciones aplicables. Estos casos son debidos a que no es posible seguir reduciendo por una rama del GSS, al representar ésta una hipótesis incorrecta. De esta forma, en el caso general de gramáticas altamente ambiguas, es inevitable que el algoritmo incluya una cierta cantidad de búsqueda/análisis especulativo, orientado a explorar estas alternativas.

2.3.9 Aspectos sutiles y limitaciones del algoritmo

En esta sección se analizan los aspectos sutiles y limitaciones más relevantes del algoritmo de Tomita: aspectos relativos al orden de elección de las acciones de reducción (sección 2.3.9.1), imposibilidad de tratar gramáticas cíclicas (sección 2.3.9.2), gramáticas con recursión a izquierdas escondida (sección 2.3.9.3), y gramáticas con reglas anulables por la derecha (sección 2.3.9.4).

2.3.9.1 Orden de elección de las acciones de reducción

Uno de los aspectos del algoritmo de Tomita que merece ser objeto de mayor análisis es el de la elección de la acción de reducción a aplicar entre las posibles reducciones aplicables para un conjunto de vértices estado activos. Efectivamente, el algoritmo elige dichas acciones arbitrariamente, del conjunto de todas las posibles acciones de reducción realizables sobre el estado actual de la GSS. Sin embargo, una interpretación *naif* de este comportamiento podría provocar que la forma final del SPPF dependiera del orden de elección. Obsérvese el caso planteado en la Figura 2.3.12:

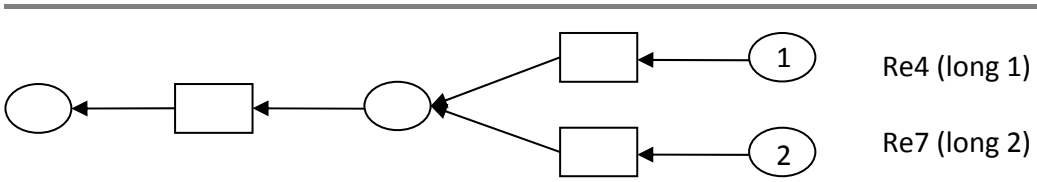


Figura 2.3.12. Estado de la GSS con dos posibles acciones de reducción aplicables.

- Si aplicamos la acción de reducción sobre el vértice estado número 1 de longitud 1, se nos genera un vértice estado número 2 ya existente. Se produce una fusión de tipo I por ambigüedad local inmediatamente. Esta fusión se muestra en la Figura 2.3.13.

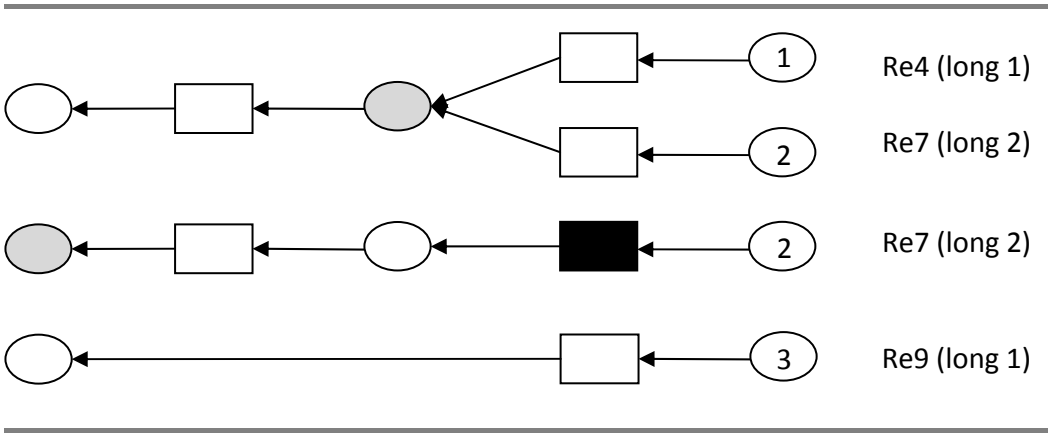


Figura 2.3.13. Estados que adopta la GSS al aplicar la acción de reducción sobre el vértice 1.

- Si por el contrario aplicamos primero la acción de reducción sobre el vértice estado 2 de longitud 2, se nos genera un vértice estado número 3. Como puede observarse en la Figura 2.3.14, si ahora aplicásemos la acción de reducción sobre el vértice estado número 1, nos volvería a generar el vértice estado número 2 que ya se había reducido. Si, a partir de este vértice regenerado, siguiéramos reduciendo, la fusión de tipo I por ambigüedad local se realizará en un lugar distinto, para un símbolo distinto, y a un nivel del bosque diferente.

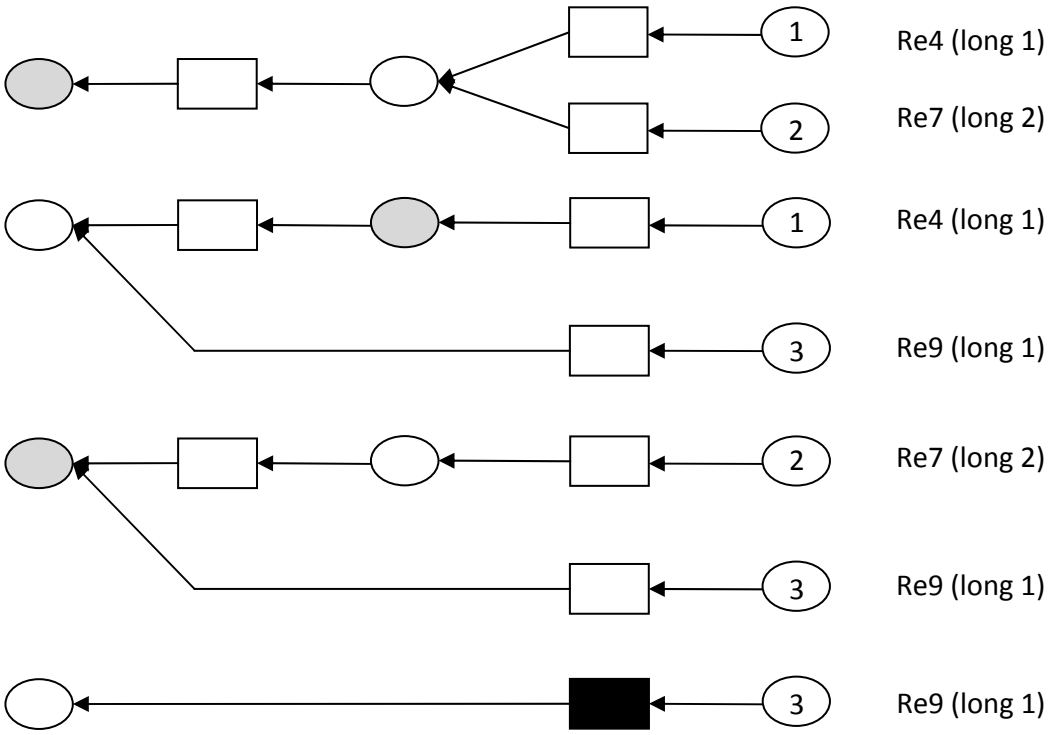


Figura 2.3.14. Estados que adopta la GSS al aplicar la acción de reducción sobre el vértice 2.

Este problema afecta a la construcción del SPPF al ser posible realizar un empaquetamiento de nodos sobre nodos ubicados a diferentes niveles en el bosque. En la Figura 2.3.13 y en la Figura 2.3.14 se ha marcado con un relleno oscuro el vértice símbolo que genera un empaquetamiento de nodos en el SPPF. Para evitar este problema, el algoritmo no reconsidera las acciones realizables sobre aquellos vértices ya generados anteriormente. De esta forma, el procesamiento llevado a cabo por el algoritmo en el caso planteado es el que se muestra en la Figura 2.3.15.

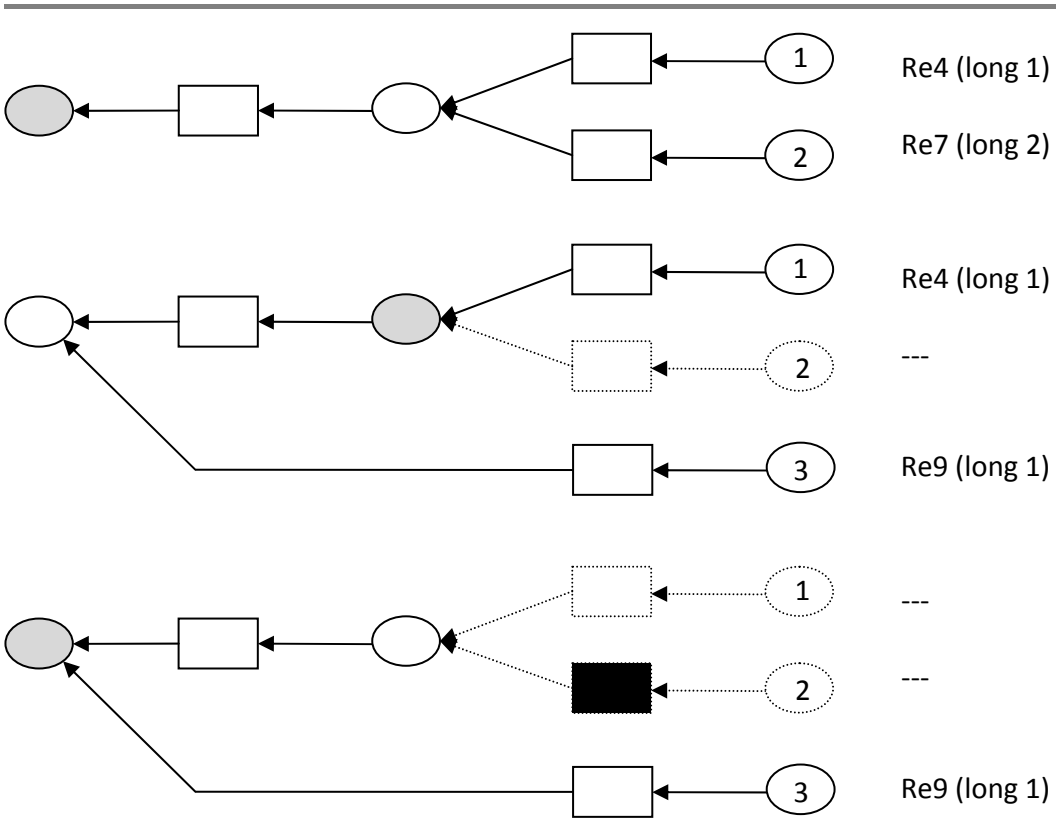


Figura 2.3.15. Estados que adopta la GSS con la implementación actual del algoritmo cuando se aplica cualquier acción de reducción.

Desafortunadamente, el hecho de no reconsiderar las acciones realizables sobre los estados regenerados induce una incompletitud en el algoritmo a la hora de reconocer gramáticas con reglas *anulables* (ver sección 2.3.9.4). De esta forma, para ciertas gramáticas, el algoritmo puede rechazar como incorrectas cadenas que sí son generadas por dichas gramáticas. Una posible forma para tratar de resolver esta incompletitud es, por supuesto, reconsiderar las acciones sobre los estados regenerados. Esta solución ofrecería, además, la ventaja de que nos permitiría podar el GSS sin necesidad de mantener guardados vértices ya procesados. Por su parte, a fin de evitar que el SPPF final dependa del orden de selección de reducciones, es posible observar que el factor que determina a qué nivel se produce el empaquetado de nodos y que incrementa el número de pasos a realizar, tiene relación directa con la longitud de la acción de reducción que se aplica. Mediante una selección de la acción de reducción a aplicar

basada en la elección de la reducción cuya longitud sea más corta, es posible resolver el problema. Desafortunadamente, aunque esta modificación funciona correctamente para casos simples de gramáticas con reglas anulables, en casos más complejos el proceso puede no terminar, entrando en un bucle infinito en el que el mismo estado se regenera y reconsidera una y otra vez.

2.3.9.2 Gramáticas cíclicas

El algoritmo de Tomita no es capaz de manejar de forma completa gramáticas cíclicas. Efectivamente, toda gramática cíclica, como la de la Figura 2.3.16, es infinitamente ambigua (en el sentido de que una misma sentencia puede tener infinitos árboles de derivación).

1. S ::= S
2. S ::= 'x'

Figura 2.3.16. Gramática cíclica.

Estos infinitos árboles podrían representarse mediante un SPPF cíclico. No obstante, el algoritmo de Tomita no construye dicho tipo de estructuras, terminando sin representar todos los árboles sintácticos (ver Figura 2.3.17).

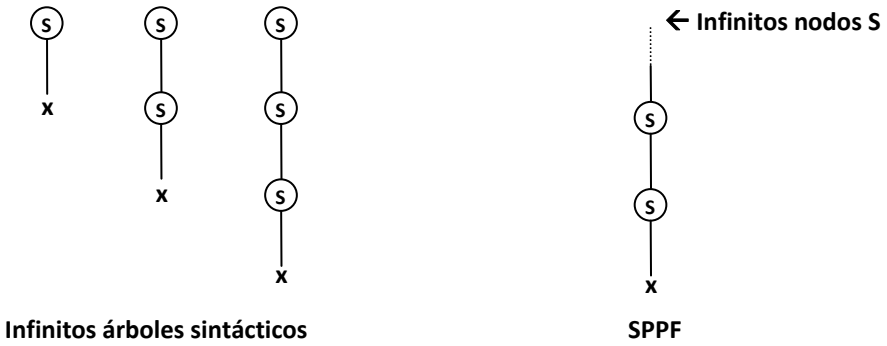


Figura 2.3.17. Arboles sintácticos y SPPF asociados con la cadena "x".

2.3.9.3 Recursión escondida a izquierdas

El algoritmo no funciona para las gramáticas que presentan *recursión escondida a izquierdas*. Una gramática presenta una recursión escondida a izquierdas si contiene una regla: A ::= αβ, donde α puede generar la cadena vacía y β una forma sentencial que comienza por A. La gramática de la Figura 2.3.18 constituye un ejemplo.

1. S ::= M S ‘b’
2. S ::= ‘a’
3. M ::= λ

Figura 2.3.18. Gramática con recursión escondida a izquierdas.

En esta gramática, y ante la cadena de entrada “ab”, el algoritmo no puede determinar cuántas veces se puede reducir por la cadena vacía antes de poder desplazar el carácter ‘a’. Esto genera un bucle infinito y la ejecución no termina.

2.3.9.4 Reglas anulables por la derecha

1. S ::= ‘a’ S A
2. S ::= λ
3. A ::= λ

Figura 2.3.19. Gramática con reglas anulables por la derecha.

Una gramática presenta reglas anulables por la derecha si contiene una regla de la forma A ::= αβ, donde β puede generar la cadena vacía. El algoritmo de Tomita termina de manera incorrecta para estos casos. La gramática de la Figura 2.3.19 contempla este caso.

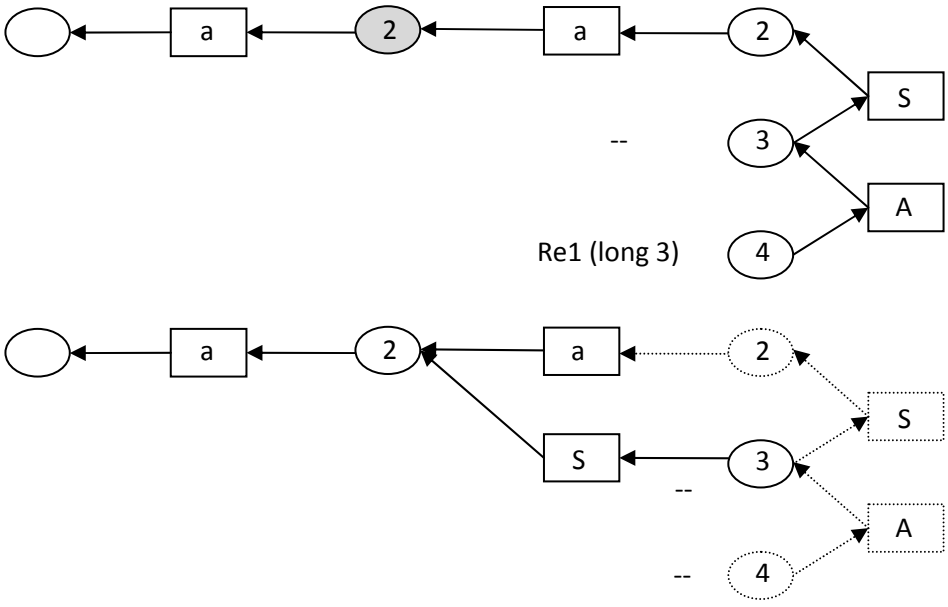


Figura 2.3.20. Estados que adopta la GSS ante reglas anulables por la derecha.

El algoritmo de Tomita termina su ejecución rechazando la cadena de entrada “aa”. Obsérvese la gramática de la Figura 2.3.19. Tras desplazar la cadena “aa”, se ejecutan dos acciones de reducción nulas, generando el vértice estado 4. Al reducir el vértice estado 4 de longitud 3 por la regla 1, se nos crea un enlace al vértice estado número 3 ya existente. La GSS

llega a adquirir un estado en el que no existen acciones aplicables sobre ninguno de sus vértices estado activos (Figura 2.3.20). El proceso termina sin éxito rechazando la cadena de entrada.

2.3.10 Mejoras a los defectos del algoritmo

Durante los últimos años, se han propuesto mejoras enfocadas a resolver algunas de las limitaciones del algoritmo propuesto originalmente por Tomita que se han descrito en el apartado anterior.

2.3.10.1 Recursión escondida a izquierdas

Farshi [Nozohoor-Farshi 1991] propone una modificación del algoritmo de Tomita que resuelve este problema. El algoritmo resultante se denomina *Correct GLR*. Sin embargo, las búsquedas introducidas en su algoritmo provocan que se comporte de manera mucho menos eficiente que en el algoritmo original.

Nederhof y Sarbo [Nederhof & Sarbo 1996] proponen una modificación del algoritmo de Tomita que resuelve el problema a base de eliminar las producciones nulas de la gramática. Su versión del algoritmo GLR se denomina *e-removal on the fly*. Para ello, se modifica el contenido de las tablas de análisis a fin anticipar o predecir el hecho de aplicar reglas de producción nulas.

2.3.10.2 Reglas anulables por la derecha

Con la herramienta de visualización desarrollada se ha probado a realizar una implementación de la modificación del algoritmo de Tomita sugerida en la sección 2.3.9.1. Dicha modificación, para el caso mostrado en la Figura 2.3.20, puede seguir reduciendo por el vértice estado 3 al volverse a generar y encolar su acción de reducción correspondiente. No obstante, y aunque termina con éxito reconociendo la cadena de entrada, no funciona para otros casos, como el mostrado en la Figura 2.3.21.

1. A ::= B
2. B ::= B ‘b’
3. B ::= ‘a’ B C
4. B ::= ‘a’ B ‘b’
5. B ::= λ
6. C ::= λ

Figura 2.3.21. Gramática con reglas anulables por la derecha.

El algoritmo de Tomita termina correctamente en este caso. Sin embargo, en la modificación ensayada del algoritmo, el proceso no termina y entra en un bucle infinito.

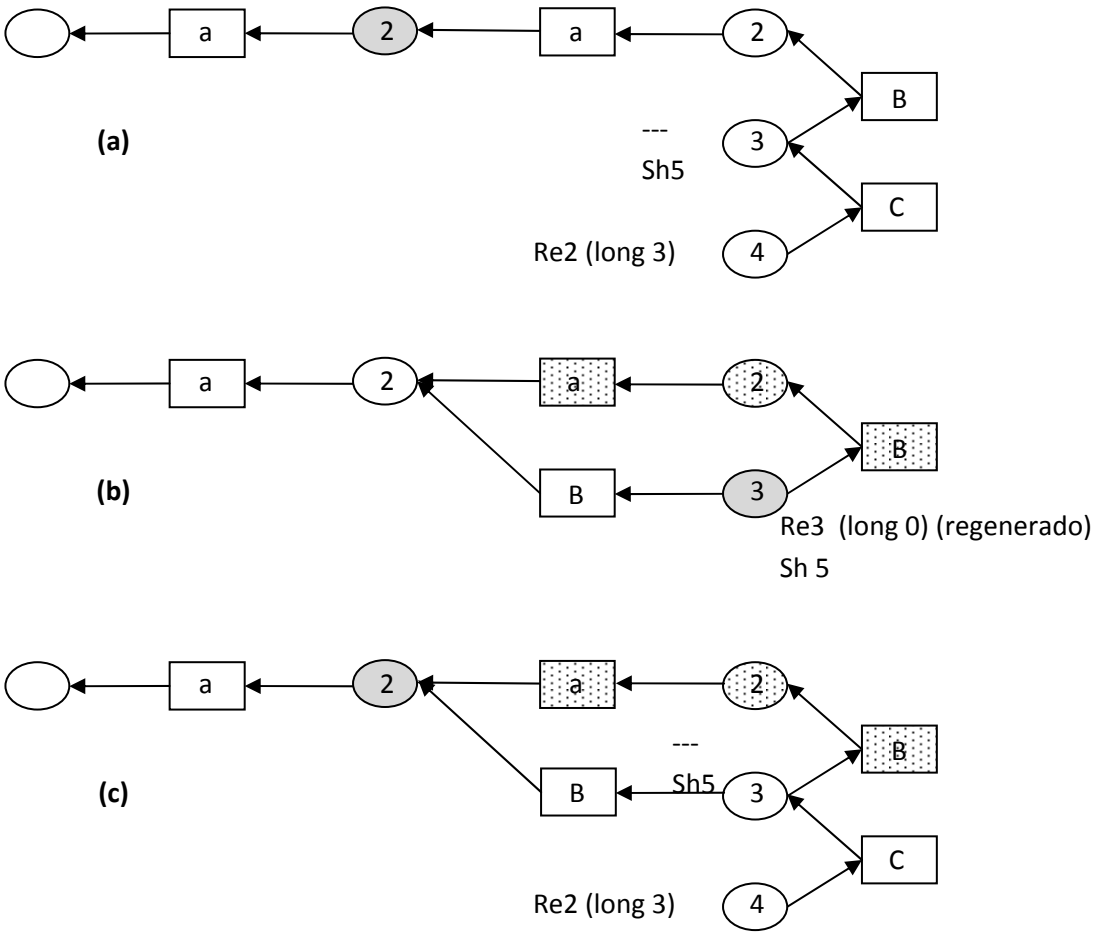


Figura 2.3.22. Estados de la GSS ante reglas anulables por la derecha mediante la modificación sugerida.

Obsérvese la Figura 2.3.22. En la parte (a) se observa cómo el camino superior no puede eliminarse tras realizar la reducción pendiente sobre el vértice estado 4 por contener el vértice estado 3 una acción de desplazamiento sin realizar. En la parte (b) se observa cómo, tras realizar la acción de reducción del vértice estado 4, se vuelven a regenerar las acciones encoladas al vértice estado 3. En la parte (c) se observa cómo ahora existe un camino adicional a reducir. Esto es incorrecto. Es más, la regeneración de las acciones de reducción en este caso producen un bucle infinito en ejecución.

Se han estudiado diferentes formas de resolverlo adicionales a las ya propuestas. Se ha concluido que la modificación del algoritmo de Tomita propuesta por Johnstone y Scott [Johnstone et al. 2004] es la que resuelve de manera más eficiente el problema. Esta modificación se denomina *Right Nulled GLR* (RNGLR), y parte de la mejora propuesta en la versión *e-removal on the fly*. Su procedimiento consiste en detectar las reglas anulables por la

derecha de la gramática y modificar las tablas de análisis para anticipar reducciones a fin de evitar realizar acciones de reducción nulas redundantes en el proceso.

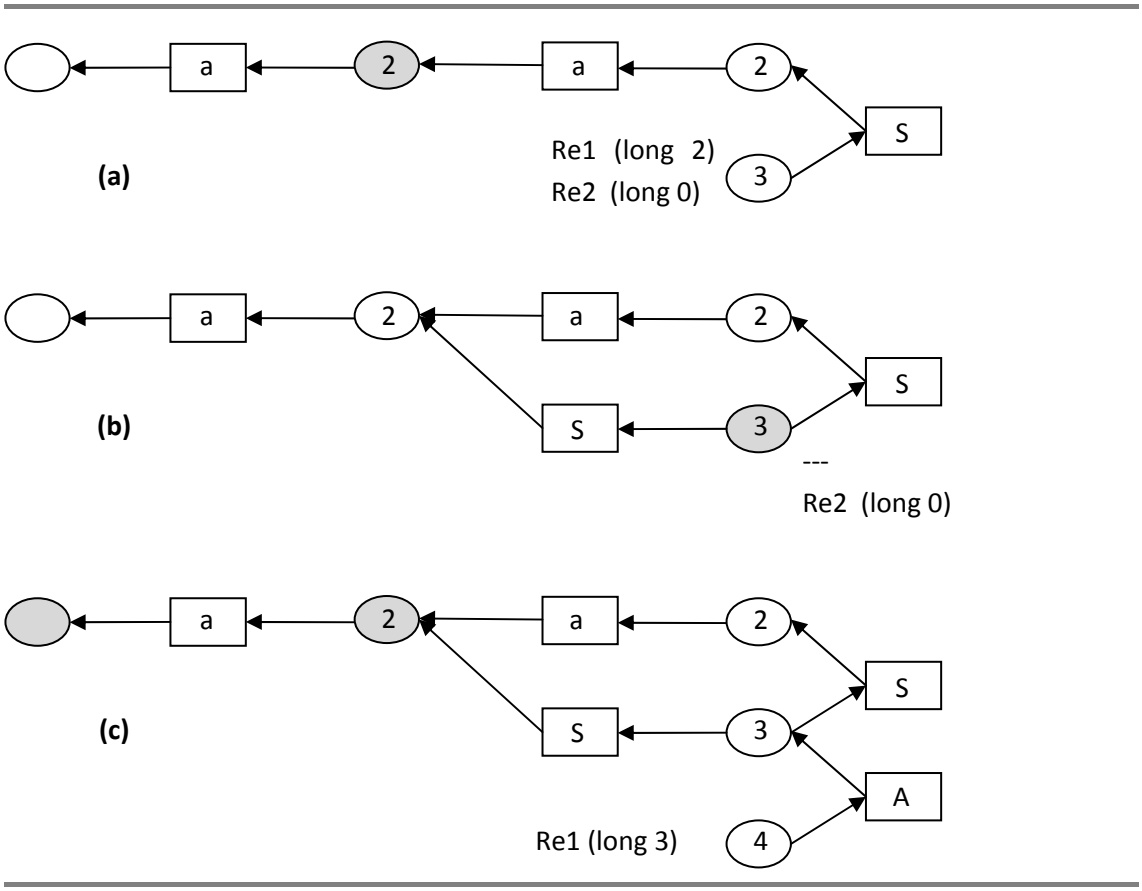


Figura 2.3.23. Estados de la GSS ante reglas anulables por la derecha mediante el método RNGLR.

Obsérvese la gramática de la Figura 2.3.19 y la Figura 2.3.23. Debido a la modificación que el algoritmo RNGLR produce en las tablas de análisis, el vértice estado 3 creado posee una acción de reducción anticipada. Tras ejecutar dicha acción, se crea el camino que ocasionaba el problema de manera anticipada. Ahora el proceso puede continuar como de costumbre y terminar con éxito aceptando la cadena de entrada.

La desventaja del algoritmo RNGLR es, no obstante, que la introducción de la citada modificación rompe con la manera original de generar el SPPF. Para la generación del bosque SPPF es necesario ahora realizar más operaciones concretas junto al uso de información de SPPFs pre-construidos debido a las reglas anulables.

2.3.10.3 Mayor compactación en los bosques

Rekers [Rekers 1992] propone una manera todavía más compacta para representar los árboles sintácticos: *Compact parse forest*. Su implementación parte del algoritmo modificado

por Farshi [Nozohoor-Farshi 1991]. Visser [Visser 1997] mejora la eficiencia del algoritmo de Rekers con un método denominado *Scannerless GLR.*

2.3.11 Algunas conclusiones

La versión del algoritmo de Tomita propuesta por Johnstone y Scott es la más eficiente que existe hasta el momento a la hora de reconocer una cadena de entrada. Sin embargo, para generar el SPPF necesita realizar más operaciones y el uso de SPPFs preconstruidos debido a cómo se tratan las reglas anulables.

El desarrollo del visualizador ha permitido realizar un estudio a fondo del comportamiento del algoritmo de Tomita. Así mismo, ha permitido analizar con mayor detalle las optimizaciones propuestas sobre el algoritmo original y comprobar que, conceptualmente, el algoritmo difiere de la implementación real al poseer un defecto producido por no dar importancia al orden en el que se eligen las acciones de reducción durante el procesamiento. La herramienta también ha permitido estudiar otras posibles implementaciones de las mejoras del algoritmo propuestas en estos últimos años. Sin embargo, la versión RNGLR se presenta como la implementación más eficiente y prometedora hasta el momento.

2.4 Gramáticas de atributos

2.4.1 El formalismo básico

En este proyecto se adoptan las gramáticas de atributos [Knuth 1968], [Paakki 1995] como mecanismo de especificación de tareas de procesamiento de documentos XML. El formalismo de las gramáticas de atributos fue propuesto por Donald E. Knuth a finales de los sesenta como un mecanismo para especificar y añadir *semántica* a los lenguajes incontextuales. Las gramáticas de atributos ofrecen un alto grado de abstracción a la hora de especificar tareas de procesamiento de lenguaje, y, al mismo tiempo, permiten la generación automática de procesadores eficientes que implementan dichas tareas.

Las gramáticas de atributos son una extensión de las gramáticas incontextuales. La sintaxis de los lenguajes se especifica en dichos términos, mientras que la semántica se expresa mediante la introducción de atributos semánticos y ecuaciones semánticas. La Figura 2.4.1 muestra un ejemplo de gramática de atributos. Tomando como ejemplo de referencia esta gramática, puede observarse que una gramática de atributos consta de:

- Una *gramática incontextual* que caracteriza la sintaxis estructural del lenguaje.
- Un conjunto de *atributos semánticos* añadidos a los símbolos de la citada gramática. Estos atributos pueden ser de dos tipos: atributos *sintetizados* (p.ej. val) y atributos *heredados* (p.ej. varsh). Los atributos toman valores en los nodos

de los árboles sintácticos impuestos por la gramática incontextual sobre las sentencias. Los valores de los atributos sintetizados representan la *semántica* de los fragmentos de sentencia que penden de los nodos (p.ej. val representa el valor de las expresiones), mientras que los de los atributos heredados representan información de contexto (p.ej. varsh representa una tabla con el valor de las variables que aparecen en las expresiones). Los atributos sintetizados en un nodo se computan a partir de los sintetizados de sus hijos y de los propios atributos heredados del nodo. Por su parte, los atributos heredados se computan a partir de los sintetizados de los hermanos y de los heredados del padre.

- Un conjunto de *ecuaciones semánticas* para cada producción. Estas ecuaciones indican cómo computar los valores de los atributos sintetizados de la cabeza y de los atributos heredados de los símbolos del cuerpo. Para ello aplican *funciones semánticas* sobre los atributos utilizados en dicho cómputo. Por ejemplo, Term.val = valorDe(Term.varsh,var.nombre) indica que para encontrar el valor del término cuando éste es una variable, es necesario buscar el valor de la variable en la tabla.

El axioma de la gramática también puede tener atributos heredados (p.ej. varsh es un atributo heredado del axioma Exp). Así mismo, los terminales también pueden tener atributos sintetizados, que se denominan *atributos léxicos* (p.ej. nombre es un atributo léxico del terminal var). Los valores de estos atributos se fijarán externamente (p.ej. los atributos léxicos se fijarán durante el análisis léxico).

Para el proceso de *evaluación de atributos* no es necesario especificar explícitamente en qué orden tienen que aplicarse las ecuaciones semánticas para encontrar los valores de los atributos en los árboles sintácticos. De esta forma, las gramáticas de atributos son mecanismos descriptivos de más alto nivel que los *esquemas de traducción* soportados por las herramientas típicas de construcción de procesadores de lenguaje (p.ej. JavaCC, ANTLR, YACC, CUP, etc.), en los que sí es necesario explicitar el orden de ejecución de las *acciones semánticas* ([Aho et al. 2007]). Por el contrario, en una gramática de atributos el orden de evaluación se deriva de las *dependencias* entre los atributos introducidas por las ecuaciones semánticas. El carácter de estas dependencias se puede analizar mediante la construcción del *grafo de dependencias entre atributos.* De esta manera, una gramática de atributos está *bien definida* siempre que los valores de los atributos puedan fijarse de manera unívoca. Ello implica la existencia de un orden de evaluación concreto para determinar los valores de los atributos. En cuanto al método de evaluación en sí, éste puede ser estático o dinámico [Ablas 1991]. Los métodos estáticos analizan la gramática durante la generación del evaluador para encontrar un orden de evaluación que funciona para cualquier sentencia. Por su parte, los métodos dinámicos deciden el orden de evaluación para cada sentencia particular (p.ej., ordenando topológicamente los nodos del grafo de dependencias asociado con el árbol de análisis sintáctico de la sentencia). En la propuesta realizada en este proyecto se adoptará un método de evaluación dinámica, ya que los métodos dinámicos aceptan una clase más amplia de gramáticas de atributos que los estáticos, aún a consta de una ligera pérdida de eficiencia.

```
Exp ::= Exp + Term
   Exp0.val = Exp1.val + Term.val
   Exp1.varsh = Exp0.varsh
   Term.varsh = Exp0.varsh

Exp ::= Term
   Exp.val = Term.val
   Term.varsh = Exp.varsh

Term ::= var
   Term.val = valorDe(Term.varsh,var.nombre)

Term ::= num
   Term.val = num.valor
```

Figura 2.4.1. Ejemplo de gramática de atributos.

2.4.2 Paradigmas de Organización y de Evaluación

El formalismo básico para las gramáticas de atributos descrito en la sección anterior adolece de falta de estructura y modularidad de las especificaciones resultantes. Para tal fin, es posible adoptar técnicas análogas a las de descomposición y estructuración de información utilizadas en los lenguajes de programación, las cuáles facilitan la construcción y el mantenimiento de los sistemas. De esta manera surgen los denominados *paradigmas organizacionales* para gramáticas de atributos [Paakki 1995], cuyo fin es mejorar los aspectos relacionados con el mantenimiento, reutilización, legibilidad y facilidad de uso de las especificaciones. En [Paakki 1995] se plantean, inicialmente, dos paradigmas básicos para la descomposición y estructuración de las gramáticas de atributos:

- *Estructuración en bloques*. Estos paradigmas se basan en el concepto de *bloque* en lenguajes de programación, de acuerdo con el cuál el lenguaje se estructura en unidades de bloques caracterizadas por contener un número arbitrario de entidades locales que, a su vez, pueden ser bloques. Las ventajas que aporta este concepto son la división lógica de los programas en una jerarquía de elementos de forma estática, y el uso de nombres locales. En relación con las gramáticas de atributos, existen dos métodos de estructuración por bloques: considerar cada símbolo no terminal como un bloque, o bien considerar las producciones como bloques sugiriendo que los atributos se sitúen a nivel local de una producción. Aunque esta conceptualización es muy primitiva, considerar los no terminales como bloques permite considerar cada no terminal como la descripción de un sublenguaje. Además, el anidamiento, permite describir la jerarquía de dichos sublenguajes. Algunos ejemplos de estos sistemas son: FNC-2 [Jourdan et al. 1991], HLP84 [Koskimies et al. 1988] y TOOLS [Koskimies & Paakki 1990].

- *Estructuración en procedimientos*: consiste en trasladar el método descendente y recursivo de los lenguajes de programación tradicionales a las gramáticas de atributos, considerando los símbolos no terminales de la gramática como

procedimientos e identificando sus atributos semánticos como parámetros del procedimiento. Con ello, cada procedimiento permite expresar el procesamiento del sublenguaje que define, en términos de otros procedimientos. Un ejemplo de estos sistemas es DEPOT2a [Grossmann et al. 1984].

Sin embargo, estos métodos de estructuración no son escalables. Debido a ello, es necesario establecer el concepto de *módulo* en el dominio de las gramáticas de atributos. La estructuración en módulos es más rica y poderosa que los bloques y procedimientos, al permitir una mejor reutilización y modificación de sus operaciones o contenidos, sin implicar cambios en otros módulos. Efectivamente, cuando el concepto se aplica al dominio de la programación, éste permite la abstracción de los datos o uso de interfaces abstractas para el acceso a los componentes, la ocultación de la información o implementación a nivel de módulo, y la compilación de módulos a partir de otros módulos de manera separada. A la hora de aplicar el concepto a las gramáticas de atributos surgen diferentes metodologías:

- *Símbolo no terminal como módulo*. Esta consideración permite utilizar la especificación de un sublenguaje en la especificación de un nuevo lenguaje mediante la importación de símbolos no terminales identificados como módulos. Los atributos del no terminal son la información accesible del módulo, quedando oculta el resto de la implementación. Algunos de estos sistemas se discuten en [Koskimies 1989].

- *Atributos semánticos como módulo*. Esta metodología consiste en considerar cada atributo, junto a todas las reglas semánticas asociadas, como unidad. Sin embargo, esto implica que cada módulo conozca aquellas producciones en las que ocurren en las reglas semánticas, lo cual no permite una buena separación entre la implementación y la ocultación de datos. Un ejemplo de estos sistemas es MAGGIE [Dueck & Cormack 1990].

- *Aspecto semántico como módulo*. Este método, propugnado en [Kastens & Waite 1994], consiste en agrupar símbolos no terminales, atributos semánticos y/o producciones que capturen un aspecto semántico (p.ej., construcción de la tabla de símbolos, comprobación de restricciones contextuales, o traducción). Un ejemplo de estos sistemas es Linguist [Declarative Systems 1992].

Como evolución de los paradigmas organizacionales basados en bloques, procedimientos y módulos, surge la aplicación de los principales conceptos de la programación orientada a objetos a las gramáticas de atributos como un refinamiento a ambos estilos, de estructuración y modularización, mediante la introducción del concepto de *clase*. Una clase combina el concepto de módulo sumado al tipo que representa la clase. Así mismo, cada instancia de la clase se define como un objeto de la clase. Estos conceptos, junto con los de herencia, generalización, polimorfismo, estado, vinculación dinámica, etc., se aplican a gramáticas de atributos de la siguiente forma:

- *Símbolo no terminal como clase*. Respetando las mismas características que posee cuando se entiende como un módulo, un no terminal se puede considerar como una clase, y sus instancias como objetos de dicha clase, cada una de las cuáles posee su propio estado. Esta metodología es consistente con el principio de procesamiento dirigido por sintaxis, ya que, durante el proceso de análisis de una sentencia, los objetos creados dinámicamente instanciado los no terminales tienen una correspondencia con los nodos del árbol de análisis sintáctico generado. TOOLS [Koskimies & Paakki 1990] es un ejemplo de estos sistemas.

- *Producción como clase*. Esta metodología permite especificar la estructura sintáctica de una producción, atributos del no terminal (correspondiente a la cabeza de la regla) y ecuaciones semánticas de la regla, permitiendo la herencia, especialización y sobreescritura en subclases, así como la introducción de propiedades tales como la vinculación dinámica en el metalenguaje. Como ejemplo de estos sistemas existe Mjølner/Orm [Hedin 1989].

De esta forma, el método orientado a objetos permite, de una manera más flexible, una mayor reutilización y extensión de las implementaciones frente a los métodos de estructuración estáticos o de modularización analizados. Como evolución de este método, surgen otro tipo de paradigmas organizaciones, como el de las gramáticas de atributos genéricas [Saraiva & Swierstra 1999], y el de las gramáticas de atributos orientadas a aspectos [Rebernak et al. 2006].

Las metodologías organizacionales analizadas se centran en el manejo de la complejidad de grandes especificaciones, dejando a un lado los aspectos relativos al proceso de evaluación. En última instancia, una especificación que aplica cualquiera de los principios indicados puede traducirse en una especificación que sigue el formalismo básico. De esta forma, dichos métodos *no* aumentan el poder expresivo de dicho formalismo básico. Como contraposición, y siguiendo de nuevo a [Paakki 95], surgen los paradigmas de evaluación que, a diferencia de los paradigmas organizacionales, si pueden aumentar dicho poder expresivo. Por ejemplo, las *gramáticas de atributos lógicas*, se basan en crear una correspondencia entre símbolos no terminales y predicados [Maluszynski 1991], o entre reglas y cláusulas de Horn [Sataluri 1998], siguiendo el paradigma de la programación lógica, mientras que las *gramáticas de atributos funcionales* realizan una correspondiente entre gramáticas de atributos y el paradigma de la programación funcional [Johnsson 1987]. Otras metodologías, por último, se centran en agilizar el proceso de evaluación, como ocurre, por ejemplo, con los métodos de *evaluación de atributos en paralelo* [Gross et al. 1989] y de *evaluación de atributos incremental* [Goldberg & Robson 1983].

2.4.3 Procesamiento de documentos XML con gramáticas de atributos

En la actualidad, existen diversos trabajos en los que se usan las gramáticas de atributos para caracterizar ciertas tareas de procesamiento de los documentos XML y de otro tipo de documentos estructurados. Sin embargo, estos trabajos se han centrado más en aspectos estructurales, como por ejemplo, realizar una asimilación de conceptos entre las gramáticas de atributos y las sintaxis EBNF subyacente a las gramáticas documentales. A este respecto, un trabajo pionero es el descrito en [Feng & Wakayama 1993], donde se presenta un sistema de transformación de documentos estructurados para dar soporte a diversos modelos documentales tales como SGML, LaTex, etc. En otros trabajos, se presenta una gramática de atributos extendida para realizar consultas sobre documentos estructurados [Neven 2005], o se utiliza un nuevo enfoque en donde las reglas semánticas se disocian de las producciones y se asocian en términos de relaciones padre-hijo [Psaila & Crespi-Reghizzi 1999], o incluso, se presenta una manera de representar los tipos de elemento de los documentos a través de símbolos no terminales de la gramática, mediante el uso de gramáticas de atributos centradas en sintaxis abstractas de los mismos [Gançarski et al. 2006].

Mientras que todos los trabajos aludidos anteriormente proponen capturar tareas de procesamiento XML con una única gramática de atributos, en [Nishimura & Nakano 2005] se describe cómo aplicar transformaciones orientadas a árboles a flujos (*streams*) XML. Para ello, el enfoque se basa en el uso de gramáticas de atributos *acopladas* [Ganzinger & Giegerich 1984] (gramáticas de atributos en las que el resultado de una son frases del lenguaje definido por la otra), consiguiendo y permitiendo la construcción de aplicaciones de procesamiento XML eficientes a partir de especificaciones de proceso orientado a árboles. En [Nakano 2004] dicha propuesta se extiende mediante un modelo teórico de ejecución orientado a árboles. La desventaja de estos enfoques es, sin embargo, la necesidad de caracterizar los flujos XML como un tipo particular de árbol degenerado, y el proceso de análisis sintáctico mediante gramáticas de atributos que transformen dichos árboles degenerados en árboles con más estructura. Mientras que el enfoque puede funcionar adecuadamente para un tipo particular de árboles, que satisface las necesidades de un tipo particular de tareas de procesamiento, difícilmente escala a árboles con estructuras arbitrarias, adaptadas a tipos arbitrarios de tareas.

A excepción de la propuesta de [Psaila & Crespi-Reghizzi 1999], que propone un modelo genérico de procesamiento de documentos XML, las propuestas anteriormente citadas se centran en tareas específicas de procesamiento, en las cuáles las gramáticas de atributos se utilizan como elementos auxiliares en términos de los cuáles formalizar dichas tareas. Existen, no obstante, algunas propuestas relacionadas que propugnan el procesamiento de documentos XML dirigido por la sintaxis, y que se basan en el uso de *esquemas de traducción* (gramáticas incontextuales con atributos semánticos y acciones semánticas intercaladas entre los símbolos de las producciones, que se ejecutan durante el análisis sintáctico [Aho et al. 2007]). Ejemplos de estas propuestas son el entorno ANTXR [Stanchfield 2009], un entorno

construido sobre la herramienta ANTLR, y RelaxNGC [Kawaguchi 2002], una extensión del *lenguaje de esquema documental* RelaxNG [Vlist 2003] utilizada para especificar esquemas de traducción y que permite la generación automática de traductores recursivos descendentes.

2.5 A modo de conclusión

Este capítulo ha revisado los aspectos conceptuales y tecnológicos más relevantes para este proyecto de Máster. En concreto:

- Se ha realizado una introducción a los lenguajes de marcado descriptivo, y en especial, al lenguaje más utilizado globalmente y que adquiere uno de los principales papeles en este proyecto: el lenguaje de marcado extensible XML. Así mismo, se han analizado las principales tecnologías disponibles para el procesamiento de dichos documentos, tales como DOM o SAX.

- Se ha realizado un estudio sobre la construcción de los analizadores ascendentes, y en especial, se ha dedicado un extenso y detallado estudio al algoritmo más utilizado y eficiente para el manejo de gramáticas que transcienden la clase LR(k): el algoritmo GLR. En dicho estudio, se ha analizado con detalle el funcionamiento del algoritmo y estudiado las distintas versiones y optimizaciones del mismo. Este análisis es esencial en este proyecto, a fin de ofrecer un soporte adecuado a múltiples vistas gramaticales de un mismo tipo de documento, que coexisten simultáneamente durante la caracterización de tareas de procesamiento de documentos XML complejas.

- Por último, se ha dedicado un apartado al formalismo básico y a los paradigmas de organización y evaluación de las gramáticas de atributos, y al uso de dichas gramáticas de atributos en lo referente al procesamiento de documentos XML, realizando un breve repaso a las aportaciones existentes sobre dicho tema.

Este proyecto pretende partir de la actual versión de XLOP, el sistema de procesamiento de documentos XML basado en gramáticas de atributos desarrollada en el 2009 [Martínez-Avilés & Temprado 2009] y [Sarasa et al. 2009d], a fin de ampliar su potencia de especificación introduciendo soporte a la *especificación modular* y a la *especificación de gramáticas de atributos multivista*, en donde adquiere especial importancia la extensión del actual lenguaje de especificación de XLOP y la integración del algoritmo GLR en el entorno para incluir dicho soporte en la generación final de las aplicaciones. Estos aspectos se analizan en los capítulos que siguen.

Capítulo 3
El Entorno XLOP. Versión Inicial, Experiencias de Uso y Extensiones para el Soporte de Especificaciones Modulares

3.1 Introducción

XLOP (*XML Language-Oriented Processing*) es un entorno que utiliza *gramáticas de atributos* para describir cómo procesar documentos XML marcados con un determinado vocabulario. De esta forma, XLOP sigue un paradigma *dirigido por lenguajes* en el desarrollo de los programas de procesamiento de documentos XML. De acuerdo con este paradigma, dichos programas se entienden como *procesadores de lenguaje*, y el proceso de desarrollo en sí se entiende como el proceso de construcción y mantenimiento de dichos procesadores. De hecho, utilizando XLOP es posible generar automáticamente tales procesadores a partir de especificaciones de alto nivel expresadas como gramáticas de atributos. XLOP es fruto de las investigaciones llevadas a cabo en el proyecto de investigación Santander/UCM PR34/07-15865 "Tecnologías de Marcado Descriptivo -XML- como base a un Proceso de Desarrollo de Software Guiado por Lenguajes". XLOP, así como alguna de sus aplicaciones y otros esfuerzos realizados con el enfoque de procesamiento XML orientado a lenguajes, se describen en distintas publicaciones [Sarasa et al. 2008, 2009a-e] y [Temprado et al. 2010a-b].

XLOP se ha diseñado para ser integrado con Java. Las *funciones semánticas* que se utilizan en las gramáticas de atributos XLOP se implementan como métodos en Java. De esta forma, XLOP ofrece una flexibilidad comparable a la de los marcos de procesamiento genéricos para XML (p.ej. SAX, DOM, STaX, etc.) y un nivel de usabilidad comparable al de enfoques específicos (p.ej. los ofrecidos por lenguajes de transformación como XSLT).

XLOP permite estructurar las aplicaciones de proceso de XML en dos capas perfectamente diferenciadas:

- Una capa de lógica específica de la aplicación, que incluye la maquinaria necesaria para soportar la funcionalidad de dicha aplicación (p.ej. un marco de aplicación para la representación interna de un tutor inteligente, un conjunto de clases para el procesamiento de metadatos, etc.).

- Una capa *lingüística* de procesamiento XML *dirigido por la sintaxis*. Esta capa se especifica como una gramática de atributos, especificación que se traduce automáticamente a una implementación ejecutable mediante un *generador* XLOP.

La conexión entre ambas capas se realiza mediante una *clase semántica*, que implementa en Java las funciones semánticas utilizadas en la gramática de atributos XLOP, y que media entre las dos capas anteriores. De esta forma, el modelo de desarrollo dirigido por lenguajes de XLOP propugna la separación explícita de estas dos capas, así como facilita el desarrollo y el mantenimiento de la capa lingüística, ya que ésta se especifica a un nivel mucho más alto que el conseguido con una implementación directa en Java o en cualquier otro lenguaje de programación.

La primera versión de XLOP (en adelante, XLOP 1.0) proporcionó soporte para especificaciones *monolíticas*: cada tarea de procesamiento se describía mediante una única gramática de atributos, situada físicamente en un único archivo. Para tareas de procesamiento complejas, sin embargo, dicha naturaleza monolítica puede no ser apropiada. Por tanto, una parte importante de los esfuerzos de investigación y desarrollo en XLOP se centran, actualmente, en dotar de modularidad a las especificaciones del sistema. En este proyecto de investigación describimos los primeros resultados obtenidos en esta línea. Efectivamente:

- Hemos dotado a XLOP de la posibilidad de fraccionar las especificaciones en múltiples *fragmentos* de gramáticas de atributos, cada uno de los cuáles puede situarse físicamente en un archivo diferente. Así mismo, con el fin de facilitar la gestión de estos fragmentos, hemos incluido en XLOP un sistema de *espacios de nombres*, que permite evitar conflictos entre los vocabularios de las especificaciones (en particular, los no terminales).

- También hemos dotado a XLOP de la posibilidad de fraccionar especificaciones de tareas complejas en múltiples *aspectos*. Cada uno de estos aspectos son gramáticas de atributos parcialmente completas, que se unen automáticamente en base a las reglas sintácticas para dar lugar a la gramática final. El enfoque es similar al seguido en el paradigma de modularización basado en *aspectos semánticos*, y, en particular, en las propuestas de [Kastens & Waite 1994].

- Por último, hemos dotado a XLOP de la posibilidad de utilizar distintas gramáticas incontextuales equivalentes en cada uno de los aspectos citados anteriormente. Con ello, cada fragmento de la especificación puede utilizar la sintaxis que más convenga al procesamiento caracterizado por el mismo. Hemos denominado *gramáticas de atributos multivista* al paradigma de especificación resultante.

Este capítulo analiza todos estos aspectos. Para ello, comienza describiendo la versión XLOP 1.0 (apartado 3.2). Seguidamente se describe, a partir de una experiencia práctica, cómo XLOP puede utilizarse para gestionar el proceso de desarrollo y evolución de aplicaciones de procesamiento XML (apartado 3.3). Por último, el capítulo termina describiendo distintas extensiones orientadas a proporcionar modularidad a las especificaciones soportadas por el sistema (apartado 3.4).

3.2 El sistema XLOP 1.0

La primera versión del entorno XLOP (XLOP 1.0) se construyó con el objetivo de ofrecer una mayor facilidad y mejor soporte a la producción y mantenimiento de las aplicaciones de procesamiento de documentos XML. XLOP 1.0 permite especificar cada tarea de procesamiento XML mediante una gramática de atributos, así como de traducir dicha gramática a código CUP, un generador de traductores ascendentes de código abierto para el lenguaje de programación Java basado en el método LALR [Appel 1997]. En este apartado se describe brevemente este sistema. Se comienza describiendo la filosofía básica subyacente a XLOP (sección 3.2.1). A continuación se presenta el lenguaje de especificación de XLOP 1.0 (sección 3.2.2). Por último, se describe el componente que traduce las especificaciones a código CUP: el *generador* (sección 3.2.3). Para más detalles sobre XLOP 1.0 puede consultarse [Martínez & Temprado 2009] y [Sarasa et al. 2009d].

3.2.1 El enfoque gramatical en XLOP

El enfoque al procesamiento de documentos XML propugnado por XLOP se basa en los dos principios siguientes:

- Una precisa caracterización de la estructura lógica del tipo de documentos XML a procesar mediante una gramática incontextual. Esta caracterización, complementaria a la descripción de la gramática documental correspondiente (la DTD o el XML esquema), permite dotar a los documentos XML de una estructura más rica, en forma de árbol de análisis sintáctico, orientada a dar soporte a la descripción del procesamiento de los documentos.

- Una clara descripción del procesamiento de los documentos XML mediante el enriquecimiento de dicha gramática con atributos y ecuaciones semánticas. Esta caracterización permite describir el procesamiento de los documentos como una computación de valores de atributos semánticos asociados a los nodos del árbol de análisis sintáctico.

Para la aplicación de este enfoque, la gramática incontextual se describe respetando la descripción estructural de los documentos XML. Los símbolos terminales pueden ser, o bien *#pcdata* para denotar el contenido textual del documento, o bien *etiquetas de apertura y cierre* propias del lenguaje descrito por la DTD. Los símbolos no terminales se definen adecuadamente en las reglas de producción, y permiten establecer de manera explícita la estructura del procesamiento que se quiere realizar sobre los documentos. El siguiente paso es describir el procesamiento en sí, mediante la incorporación de semántica al lenguaje de marcado, definido por la gramática incontextual. A las producciones de la gramática se le asocian funciones y ecuaciones semánticas, en donde los atributos semánticos pueden ser:

- Atributos de elementos XML, atributo léxico *text* para los símbolos *#pcdata* o los mismos atributos léxicos definidos en la DTD para las *etiquetas de apertura.*

- Atributos sintetizados y heredados correspondientes a los símbolos no terminales definidos adecuadamente según el tipo de procesamiento que se quiere realizar.

A continuación se muestra un ejemplo de aplicación de este enfoque sobre un tipo de documentos XML muy sencillo, orientado a codificar expresiones aritméticas. La DTD de este tipo de documentos se muestra en la Figura 3.2.1a.

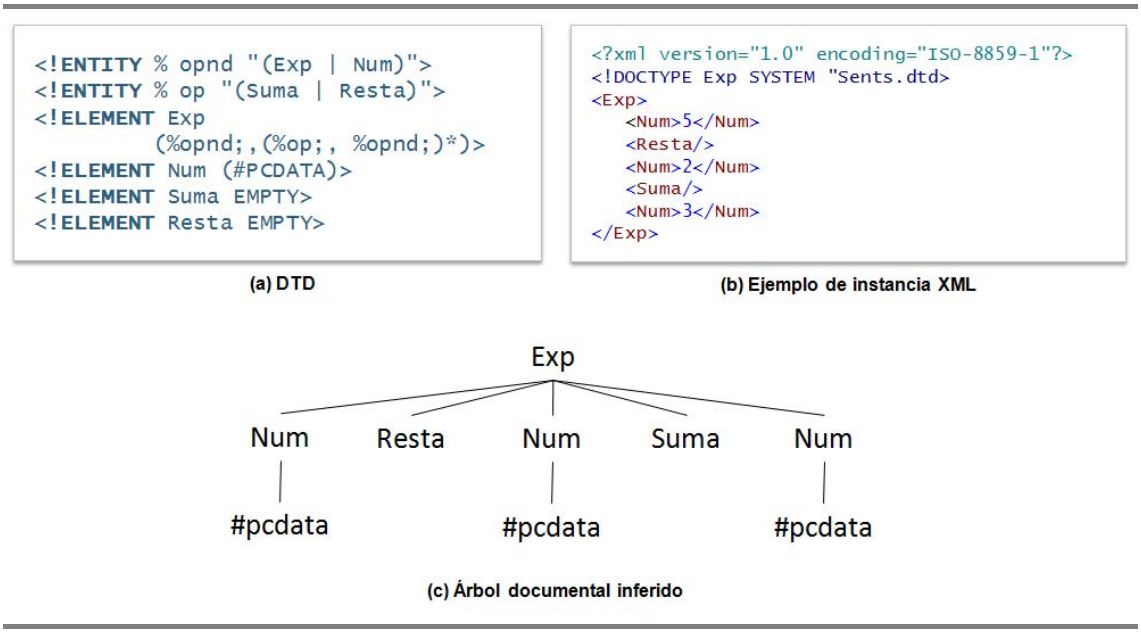


Figura 3.2.1. Ejemplo de DTD, documento XML y árbol documental inferido.

La Figura 3.2.1b muestra un ejemplo de documento XML que codifica la expresión “5 - 2 + 3”. La Figura 3.2.1c muestra el correspondiente árbol documental. Como puede observarse, la estructura de este árbol es una estructura plana, y no refleja propiedades estructurales más finas, que pueden resultar esenciales para llevar a cabo un procesamiento correcto de los documentos (en este caso, la asociatividad de los operadores).

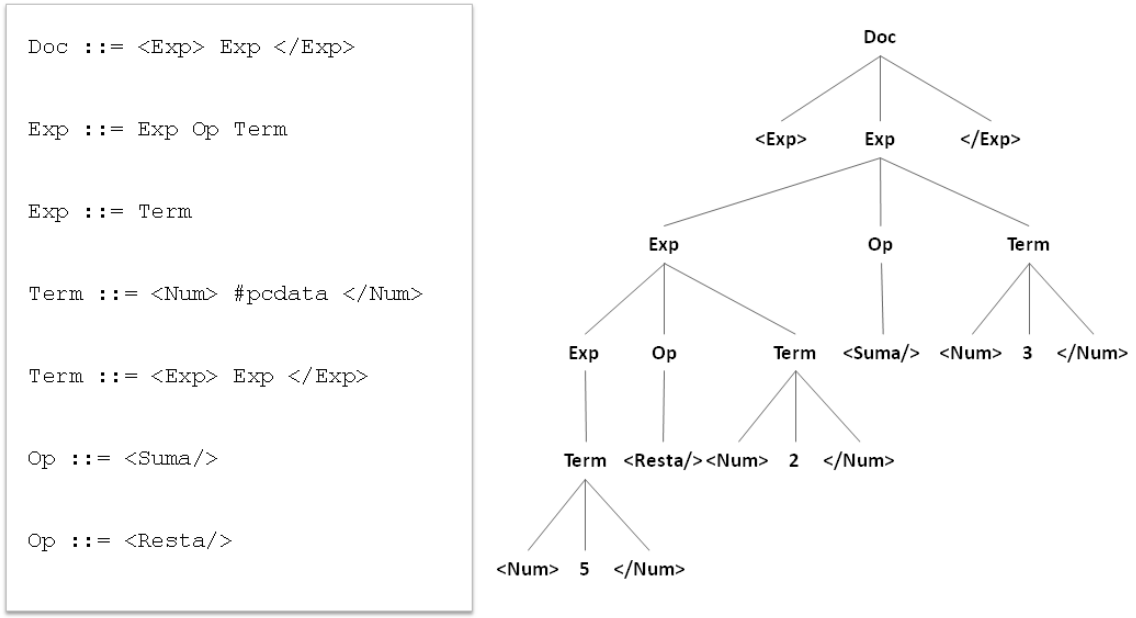


Figura 3.2.2. Caracterización de la estructura lógica de los documentos XML de la Figura 3.2.1 con una gramática incontextual.

Asumiendo que se desea implementar una tarea de procesamiento de los documentos de expresiones aritméticas consistente en encontrar el valor de las expresiones codificadas, es interesante disponer de una caracterización explícita de las asociatividades de los operadores. Esto se consigue mediante la gramática incontextual de la Figura 3.2.2a. La Figura 3.2.2b muestra el árbol de análisis sintáctico impuesto por dicha gramática sobre el documento de la Figura 3.2.1b. En contraposición con el árbol documental de la Figura 3.2.1c, dicho árbol de análisis sintáctico sí refleja ya propiedades estructurales básicas para este procesamiento, como la asociatividad de los operadores (ahora la expresión se procesa como "(5 - 3) + 3"). Esto permite que el procesamiento pueda especificarse siguiendo un estilo dirigido por la sintaxis, y, en particular, utilizando gramáticas de atributos. La Figura 3.2.3a muestra la caracterización de este procesamiento mediante una gramática de atributos. La Figura 3.2.3b esquematiza cómo se lleva cabo, conceptualmente, el cómputo de los valores requerido por dicho procesamiento.

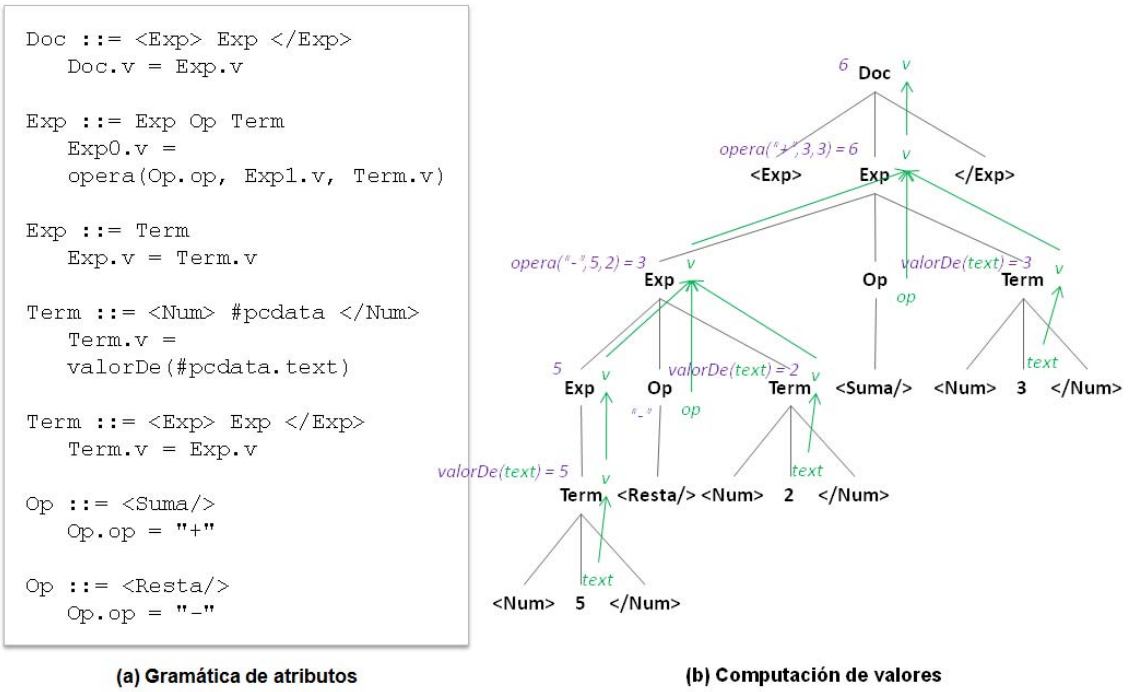


Figura 3.2.3. Descripción del procesamiento de los documentos XML de la Figura 3.2.1 con gramáticas de atributos.

3.2.2 El lenguaje de especificación

En base a las consideraciones realizadas en la sección anterior, la creación de una aplicación XLOP comienza con la especificación de una gramática de atributos: nos referiremos a este tipo de gramáticas como *gramáticas XLOP*. La Figura 3.2.4 muestra un fragmento de gramática XLOP, escrita en el lenguaje de especificación del sistema.

```
Answer ::= <Answer> <Response> #pcdata </Response> Feedbacks </Answer> {
   tutorialCreatedh of Feedbacks = tutorialCreatedh of Answer
   answh of Feedbacks =
           after(tutorialCreatedh of Answer, newAnswer(text of #pcdata))
   answ of Answer = answ of Feedbacks
}

Feedbacks ::= Feedbacks Feedback {
  tutorialCreatedh of Feedbacks(1) = tutorialCreatedh of Feedbacks(0)
  tutorialCreatedh of Feedback = tutorialCreatedh of Feedbacks(0)
  answh of Feedbacks(1) = answh of Feedbacks(0)
  answ of Feedbacks(0) =
           addFeedback(answ of Feedbacks(1), elem of Feedback)
}

Feedbacks ::= Feedback {
   tutorialCreatedh of Feedback = tutorialCreatedh of Feedbacks
   answ of Feedbacks = addFeedback(answh of Feedbacks, elem of Feedback)
}
```

Figura 3.2.4. Ejemplo de fragmento de gramática XLOP. Definición sintáctica sintáctica (en azul) y semántica (en verde) de una regla de la gramática.

En una gramática XLOP, cada regla sintáctica está compuesta por dos partes: una parte izquierda o *cabecera de la regla* y una parte derecha *o cuerpo de la regla*. Como es habitual, dichas partes se especifican separadas por "::=". Los distintos elementos que pueden formar parte de la definición sintáctica de una regla son:

- *No terminal*: Los no terminales se especifican mediante nombres siguiendo la misma sintaxis de un identificador Java ([Gosling et al. 2005]).

- *Elemento XML*: Son elementos pertenecientes al lenguaje de marcado XML y tienen un formato análogo al que presentan en las instancias de documentos XML, pero sin atributos (ver [Bray et al. 2008] para más detalles). Pueden ser de tres tipos: *etiqueta de apertura* (p.ej. <Desc>), *etiqueta de cierre* (p.ej. </Desc>) o *elemento vacío* (p.ej. <Desc/>). Cada elemento vacío es equivalente a la correspondiente etiqueta de apertura, inmediatamente seguida de la correspondiente etiqueta de cierre (p.ej., <Desc/> es equivalente a <Desc></Desc>).

- *Texto XML*: Denotan fragmentos de texto en el documento XML. Se declaran mediante "#pcdata".

La cabeza de la regla únicamente puede estar formada por un elemento *No terminal*. El cuerpo de la regla, sin embargo, se compone de una secuencia de elementos de cualquiera de los tipos enumerados. Las reglas de producción vacías, o reglas *lambda*, simplemente se declaran sin especificar elemento alguno en el cuerpo de la regla.

La definición semántica de una regla se compone de un conjunto de ecuaciones y se especifica entre llaves ("{" y "}"). En una ecuación se distinguen dos partes: una parte izquierda o *referencia a atributo* de un no terminal y una parte derecha o *expresión semántica*, separadas por "=". Las expresiones semánticas pueden ser de tres tipos:

- *Referencia a atributo*: Expresa el atributo perteneciente a un símbolo específico de la definición sintáctica de la regla. Los atributos, al igual que los no terminales, se especifican mediante nombres siguiendo la misma sintaxis que un identificador Java. La referencia a atributo se especifica mediante: un atributo, seguido de "of", seguido de un símbolo y, opcionalmente, seguido de un número de ocurrencia entre paréntesis. Es necesario especificar un número de ocurrencia cuando existen varios símbolos con el mismo nombre del de la referencia en la definición sintáctica de la regla. El número de ocurrencia identifica el símbolo mediante un número (comenzando en cero) y contando el número de veces que aparece repetido el elemento a la izquierda de su posición (véase, por ejemplo, la cuarta regla semántica de la Figura 3.2.4). Sólo es necesario especificar un número de ocurrencia cuando se definen varios elementos iguales en la definición sintáctica de una regla. Esto permite identificar al elemento deseado. No especificar un número de ocurrencia equivale a referirse a su primera aparición.

- *Valor*: Se especifican entre puntos "·" y permiten introducir valores concretos en las ecuaciones (por ejemplo, ·"Respuesta 0"·, valor que se reconoce como tipo String de Java al haberse introducido las comillas).

- *Expresión funcional*: Representa la aplicación de una *función semántica* a una secuencia de *argumentos*, cada uno de los cuáles es, a su vez, una expresión semántica. Las funciones semánticas refieren a métodos implementados en una clase Java, que se denomina la *clase semántica* de la aplicación. La expresión en sí se especifica de la siguiente manera: nombre de la función y, entre paréntesis, los distintos argumentos separados por coma. El tipo y número de argumentos debe coincidir obligatoriamente con el tipo y número de argumentos del método al que refiere la función en la clase semántica. Véanse, a modo de ejemplo, las ecuaciones de la Figura 3.2.4.

En cuanto al *axioma* de la gramática, se toma la cabeza de la primera regla de la gramática en el fichero de especificación. También, es posible escribir comentarios de dos maneras: comentario de línea mediante "//" o, comentario de párrafo entre las llaves "/*" y "*/". Hay que tener en cuenta que la definición semántica de una regla siempre debe estar contenida entre llaves "{" y "}". Es más, estas llaves siempre deben escribirse aunque no se inserte ninguna ecuación.

En el lenguaje de especificación de XLOP, no es necesario indicar explícitamente qué atributos son heredados y cuáles son sintetizados, debido a que esta información se deduce del uso de los atributos en la gramática. Más concretamente, los atributos heredados se determinan de la siguiente manera:

- Si para un atributo de un no terminal que pertenece al cuerpo de una regla de producción se define una ecuación de asignación a dicho atributo, entonces el atributo se considerará atributo heredado para dicho no terminal.

- Si un atributo de un no terminal que pertenece a la cabeza de una regla de producción se emplea en las partes derechas de las ecuaciones, el atributo se considerará atributo heredado para dicho no terminal.

Los atributos sintetizados se determinan en base a las siguientes reglas:

- Si para un atributo de un no terminal que pertenece a la cabeza de una regla de producción se define una ecuación de asignación a dicho atributo, entonces el atributo se considerará sintetizado para dicho no terminal.

- Si un atributo de no terminal que pertenece al cuerpo de una regla de producción se emplea en las partes derechas de las ecuaciones, el atributo se considerará atributo sintetizado para dicho no terminal.

La estructura de atributos sintetizados y heredados debe respetar las siguientes restricciones contextuales (de semántica estática) adicionales:

- Un atributo sintetizado perteneciente a un no terminal debe declararse mediante una ecuación en todas las reglas de producción cuya cabeza se corresponda con el no terminal.
- En los atributos de la forma *a of r* o *a of r*(*n*), la referencia sintáctica *r* debe existir en la producción. Además, si el atributo es de la forma *a* of r(*n*), debe existir al menos *n+1* ocurrencias de *r* en dicha producción (esto es porque el número de orden de la primera ocurrencia es 0).
- Los atributos heredados y sintetizados de cada no terminal han de ser disjuntos.
- No tiene sentido asignar más de una ecuación al mismo atributo.
- El único atributo léxico de #pcdata es "text", a través del cual se obtiene el contenido.
- Sólo se pueden asignar valores a atributos de no terminales en una ecuación.

La Figura 3.2.5 resume la sintaxis del lenguaje de especificación de XLOP 1.0.

SpecXLOP ::= {*Regla*}+

Regla ::= *NoTerminal* '::=' { *ElementoSintactico* }* '{' { *Ecuacion* }* '}'

ElementoSintactico ::= *NoTerminal* | **#pcdata** | *Elemento*XML

*Elemento*XML ::= *EtiquetaApertura* { *ElementoSintactico* }* *EtiquetaCierre* |
EtiquetaElmVacio

Ecuacion ::= *ReferenciaAtributo* '=' *ExpresionSemantica*

ExpresionSemantica ::= *Funcion* '(' (*ExpresionSemantica* { , *ExpresionSemantica* }*)? ')' |
ReferenciaAtributo | *ValorLiteral*

ReferenciaAtributo ::= *Atributo* **of** (*NoTerminal* | **#pcdata** | *EtiquetaApertura*)
('(' *NumeroOcurrencia* ')')?

Figura 3.2.5. Gramática EBNF de alto nivel del lenguaje de especificación de XLOP 1.0.

3.2.3 El Generador

XLOP dispone de un generador que, a partir de una gramática XLOP, produce una especificación de CUP, cuya compilación mediante CUP, genera la implementación final en Java de la aplicación XML. Por otro lado, permite realizar optimizaciones sobre la gramática de partida, produciendo especificaciones optimizadas cuya compilación, genera implementaciones de la aplicación de procesamiento final más eficientes en tiempo y consumo de memoria.

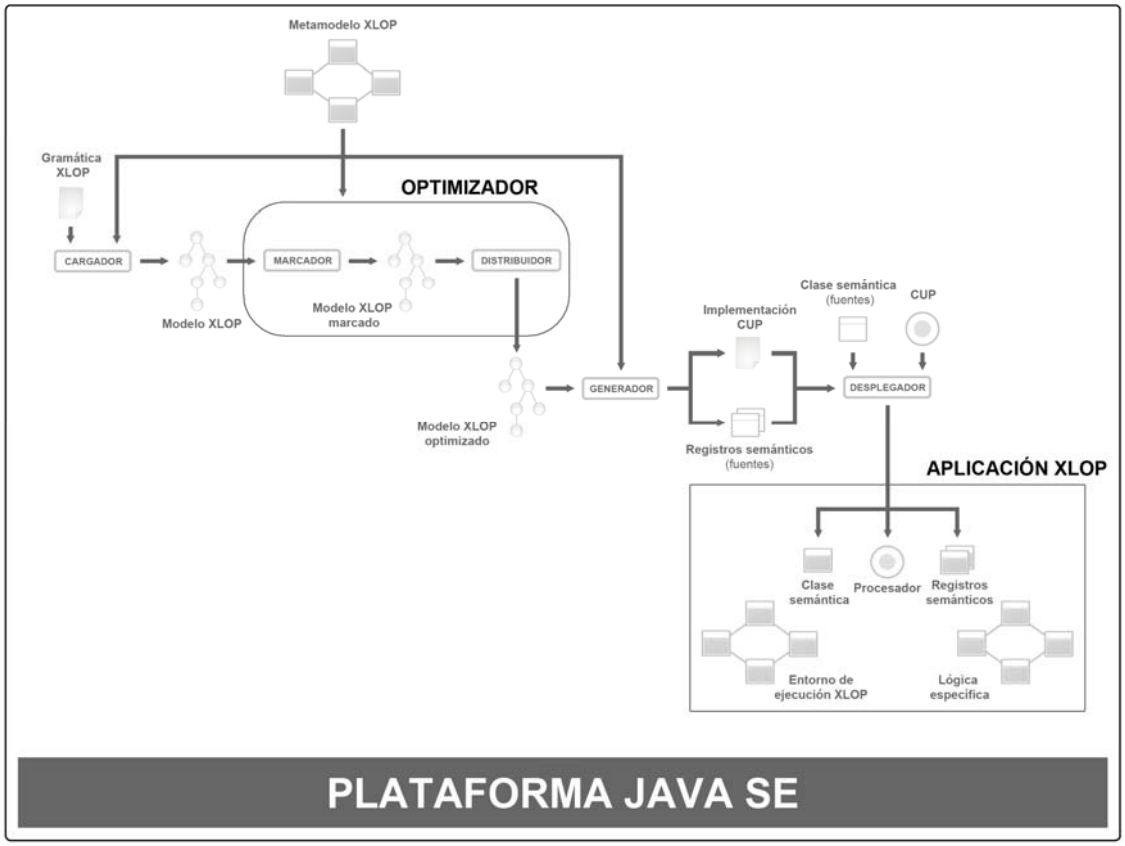


Figura 3.2.6. La arquitectura del entorno XLOP.

La Figura 3.2.6 muestra la arquitectura del entorno XLOP. El proceso de generación parte de la gramática XLOP. Mediante el módulo *Cargador*, se realiza el análisis de dicha gramática. De esta forma, el módulo *Cargador* comprueba que el archivo de especificación es correcto y que la gramática cumple las restricciones contextuales establecidas, generando un *modelo XLOP*, regido por el *Metamodelo XLOP* (conjunto de clases Java para representar internamente las gramáticas XLOP), durante el análisis de la *gramática XLOP*. El *modelo XLOP* contiene, de esta forma, toda la información extraíble de la *gramática XLOP*, estructurada de una manera especial para permitir un eficiente tratamiento y procesamiento de dicha información.

Durante el proceso de generación, se puede optar por realizar o no una optimización de la gramática original. Para dicho fin, se introduce el *Módulo Optimizador*. Dicho módulo, toma como entrada el *modelo XLOP* y genera un *modelo XLOP optimizado*. Este último modelo optimizado representa una gramática que es una mejora de la gramática original, en el sentido de incluir nuevas reglas de producción vacías (asociadas a símbolos *Marcadores* [Aho et al. 2007], que permiten almacenar atributos heredados de distintos símbolos de la gramática, así como activar ecuaciones para el cálculo de los valores de dichos atributos).

El módulo *Generador*, produce un archivo escrito en el lenguaje de especificación CUP (*Implementación CUP*) de la gramática XLOP o, de la gramática XLOP optimizada. Para ello opera sobre el *modelo XLOP* o el *modelo XLOP optimizado,* según la opción elegida. Por otro

lado, el *Generador* produce una serie de clases Java, los *Registros semánticos*, que definen la forma de gestionar el acceso y la obtención de los valores de los atributos asociados a los distintos símbolos no terminales de la gramática (incluidos marcadores, en caso del modelo optimizado).

El módulo *Desplegador* se encarga de realizar la compilación de la *Implementación CUP* mediante el compilador de CUP, y de aportar las clases necesarias que completan la implementación del procesador. El paquete de clases más importante que aporta, es el *Entorno de ejecución.* Dicho entorno permite el análisis secuencial de los archivos XML a través de una implementación específica basada en un parser SAX, y transmite por peticiones, en forma de *tokens*, los elementos XML analizados al parser CUP. Además, contiene las clases e interfaces utilizadas para soportar la evaluación: la interfaz *Continuation* y la clase *Attribute* (Figura 3.2.7).

```
public interface Continuation {
   void next();
}


public class Attribute  {
  private Object value;
  private Stack pendingConts;

  public Attribute() {
    value = null;
    pendingConts = new Stack();
  }
  public Object get() {
    return value;
  }
  public void set(Object value) {
   this.value = value;
   while (! pendingConts.empty())
     ((Continuation)pendingConts.pop()).next();
  }
 public void whenAvaliable(Continuation aux) {
    if (get() != null) aux.next();
    else pendingConts.push(aux);
  }
}
```

Figura 3.2.7. Clase Attribute e Interfaz Continuation.

La evaluación de atributos en XLOP sigue un patrón de ejecución retardada. Las operaciones que no son posibles de realizar en un determinado momento debido a la indisponibilidad del valor de un atributo, se apilan como *continuaciones* en una pila de atributos pendientes del mismo (véase método *whenAvailable* de la clase *Attribute*). Cuando el valor del atributo llega a estar disponible, éste ejecuta todas las operaciones pendientes del mismo (véase método *set* de *Attribute*). Nótese que las operaciones que se desean ejecutar en

espera a que el dato se encuentre disponible se codifican bajo la implementación del método *next()* de la interfaz *Continuation*.

La Figura 3.2.8 muestra un fragmento de uso de este patrón, en la parte de acciones de una regla CUP generada. Nótese que el método *get*() de la clase Attribute que proporciona el dato se invoca finalmente cuando puede asegurarse que el dato ya está disponible; en otro caso, la implementación del correspondiente método *next()* crea una nueva continuación y la encola en la cola de espera del siguiente atributo. Esto permite ejecutar, de manera correcta y eficiente, las operaciones que quedan pendientes del cálculo de un dato en el momento de su ejecución.

```
ListaI ::= ElmI: ElmI0
{:
   RESULT = new SemanticListaI();
   final SemanticElmI a1 = ElmI0;
   final SemanticListaI a0 = RESULT;

   a0.decsh.whenAvaliable (
      new Continuation() {
         public void next() {
            a1.decsh.set(a0.decsh.get());
         }
      }
   );

   a1.resul.whenAvaliable (
      new Continuation() {
         public void next() {
            a0.resul.set(a1.resul.get());
         }
      }
   );
:};
```

Figura 3.2.8. Aplicación del patrón de ejecución retardada.

Por último, es importante indicar también que el módulo *Desplegador* integra todos los archivos necesarios, incluyendo la *clase semántica* y la *lógica especifica de la aplicación*, permitiendo generar un paquete con la implementación final de la aplicación lista para su uso.

3.3 Desarrollo de aplicaciones de procesamiento XML con XLOP

En este apartado se describe brevemente, con un caso práctico, el modelo de producción y mantenimiento de aplicaciones XML con XLOP. El caso práctico elegido es <e-Tutor>, un sistema experimental para el desarrollo de tutores socráticos en el ámbito del e-Learning [Sierra et al. 2008b]. Los contenidos de esta sección están basados en [Temprado et al. 2010a].

El desarrollo de aplicaciones de procesamiento XML con XLOP es un caso particular del desarrollo dirigido por *lenguajes específicos de dominio* [Deursen et al. 2000; Mernik et al. 2005] de herramientas de autoría presentado en la Figura 3.3.1. De acuerdo con este modelo,

el desarrollo de las aplicaciones comienza con la conceptualización del lenguaje específico en el que se expresarán las descripciones producidas mediante la herramienta.

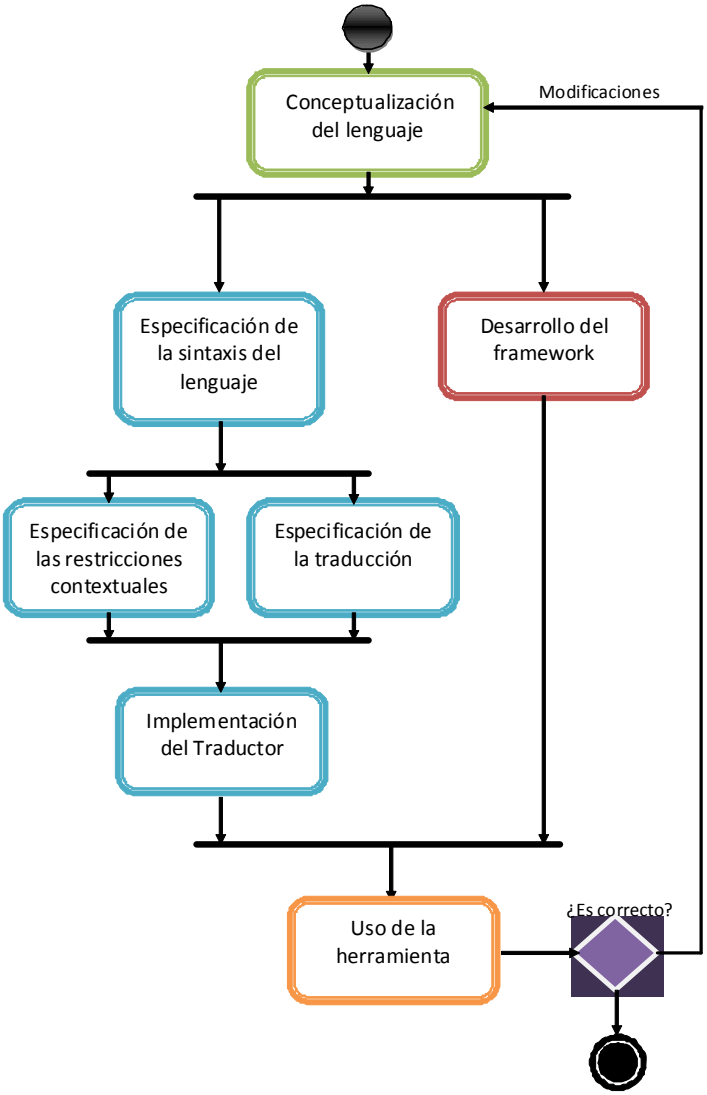


Figura 3.3.1. Modelo de desarrollo para la producción y evolución de aplicaciones de procesamiento XML con XLOP.

Una vez conceptualizado informalmente dicho lenguaje, el proceso se divide en dos etapas paralelas:

- La construcción del traductor. Este proceso comienza con la especificación de la sintaxis del lenguaje, mediante una gramática incontextual. Aquellas restricciones contextuales que deben satisfacerse en las especificaciones, y que no capta la gramática incontextual, se establecen mediante el uso de atributos y ecuaciones semánticas. De la misma forma, se describe cómo mapear las descripciones en el lenguaje en un *framework* específico asociado a la herramienta, cuyas instancias

caracterizan los objetos descritos por la misma. Esta construcción puede realizarse manualmente, o con ayuda de una metaherramienta, que genere automáticamente la herramienta a partir de las especificaciones basadas en gramáticas de atributos.

- El desarrollo del *framework* citado anteriormente.

El desarrollo de aplicaciones de procesamiento XML con XLOP adquiere las ventajas de este modelo de desarrollo. Como bien sugiere la Figura 3.3.1, este modelo es iterativo e incremental, y presenta una separación explícita del proceso de desarrollo: por un lado, la construcción del traductor, y por el otro, la construcción del framework. Esta separación permite que un cambio de funcionalidad en el framework, ya sea una corrección de errores, mejora o ampliación de funcionalidad, no implique cambios en el traductor. Así mismo, también permite que un cambio en la especificación del lenguaje no implique cambios en el framework. Con todo ello, se consigue una mejora de la productividad en el desarrollo de las aplicaciones, y una mejora sustancial en el mantenimiento y evolución de las mismas.

Estas ventajas se han experimentado con XLOP durante el desarrollo de <e-Tutor>, sistema basado en los trabajos de Alfred Bork y su equipo durante la década de los ochenta del siglo pasado [Bork 1985; Ibrahim 1989]. Como primera versión de <e-Tutor>, siguiendo el modelo de desarrollo planteado, el primer paso realizado fue la conceptualización del lenguaje. Dicha conceptualización se formalizó mediante una DTD (Figura 3.3.2a). Inicialmente, el lenguaje sólo admitían tres tipos distintos de elementos: elementos para mostrar texto, elementos para mostrar imágenes, y elementos para poder formular preguntas. En cuanto a la construcción del framework, se incorporó soporte a las citadas funcionalidades. Así mismo, se caracterizó el procesamiento de los documentos XML con una gramática XLOP (Figura 3.3.2b), utilizando la correspondiente clase semántica para introducir las operaciones de instanciación del framework. Gracias al entorno XLOP, la implementación del traductor se generó de manera automática a partir de la especificación. Dicho traductor permitió la ejecución de tutoriales como los mostrados en la Figura 3.3.2c. Posteriormente, la manera de presentar texto, imágenes, o incluso de realizar preguntas, se mejoró en el framework sin tener que cambiar la especificación XLOP (Figura 3.3.2d). De la misma manera, la gramática XLOP se modificó a fin de describir un procesamiento más mantenible, manejando explícitamente estado en las instancias de la clase semántica (Figura 3.3.2e). Del mismo modo, esto no supuso cambios en el framework.

```
...
<!ENTITY % tElement "(Text|Image)">
<!ENTITY % qPoint "(QuestionPoint)">
...
<!ELEMENT Problem (%tElement;*, %qPoint;)>
...
```

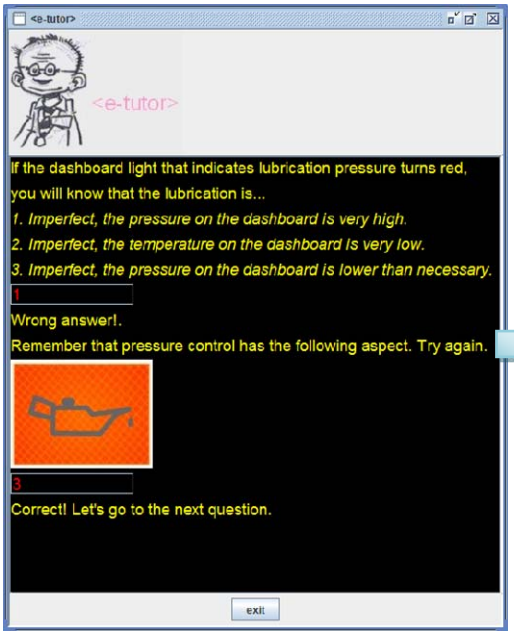
(a) Fragmento DTD de los tutoriales de <e-Tutor>

```
...
Problems ::= Problems Problem {
  unlinkedTutorial of Problems(0) =
     addNewProblem(unlinkedTutorial of Problems(1),
                   id of Problem, elem of Problem)
  unlinkedTutorial_i of Problems(1) =
                   unlinkedTutorial_i of Problems(0)
  unlinkedTutorial_i of Problem =
                   finalTutorial of Problems(1)
  finalTutorial of Problems(0) =
                   finalTutorial of Problem
}
...
```

(b) Fragmento de gramática XLOP

```
...
Problems ::= Problems Problem {
   tutorialCreated_i of Problems(1) =
       tutorialCreated_i of Problems(0)
   tutorialCreated_i of Problem =
       tutorialCreated_i of Problems(0)
   problemsCreated of Problems(0) = after(
       problemsCreated of Problems(1),
       newProblem(id of Problem, elem of Problem))
}
...
```

(e) Gramática XLOP mejorada



(c) Aplicación en ejecución con framework inicial

(d) Aplicación en ejecución con framework mejorado

Figura 3.3.2. Iteraciones en la construcción y mejora de <e-Tutor>.

A medida que surgía la necesidad de mejorar la complejidad y diversidad de los tutoriales, surgió la necesidad de extender la funcionalidad de <e-Tutor>. Para ello, se decidió dar a <e-Tutor> la capacidad de reproducir vídeos, flash, videos de Youtube y poder formular preguntas interactivas. Estas mejoras implicaron un cambio en el lenguaje. La Figura 3.3.3a presenta un fragmento de la DTD final de <e-Tutor>. Para adoptar dichos cambios, paralelamente se incluyó soporte a dichas funcionalidades en el framework y se actualizó la gramática XLOP para

describir el procesamiento necesario requerido para estos nuevos elementos (Figura 3.3.3b). Gracias al entorno XLOP, la implementación final de la aplicación se reorganizó de manera inmediata. La Figura 3.3.3c muestra el resultado.

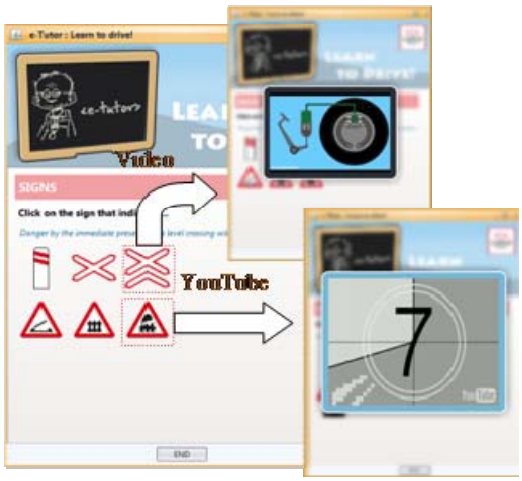
```
...
<!ENTITY % tElement "(Text|Image |
                      video | flash | youTube)">
<!ENTITY % qPoint "(QuestionPoint |
                      imageQuestionPoint)">
...
<!ELEMENT Problem (%tElement;*, %qPoint;)>
...
```

(a) Fragmento DTD de los tutoriales de <e-Tutor> evolucionado

```
...
Problems ::= Problems Problem {
   tutorialCreated_i of Problems(1) =
       tutorialCreated_i of Problems(0)
   tutorialCreated_i of Problem =
       tutorialCreated_i of Problems(0)
   problemsCreated of Problems(0) = after(
       problemsCreated of Problems(1),
       newProblem(id of Problem, elem of Problem))
}
...

...
Element ::= <YouTube/> {
  elem of Element = after(
   tutorialCreated_i of Element,
    newYouTube(path of <YouTube>,
        dialog of <YouTube>,
           newColor(dcolor of <YouTube>),
           continueBtn of <YouTube>,
        cleanScreen of <YouTube>))
}
...
```

(b) Gramática XLOP con soporte a los nuevos elementos



(c) Framework con soporte a las nuevas funcionalidades

Figura 3.3.3. Evolución de <e-Tutor>.

3.4 Incorporación de mecanismos de modularidad a XLOP

La primera versión de XLOP sólo era capaz de manejar especificaciones monolíticas: gramáticas que, en un único fichero, amalgamaban todas las producciones, atributos y ecuaciones semánticas. Con ello se desaprovechaba el carácter modular intrínseco de las gramáticas de atributos. Con el fin de subsanar este aspecto, se ha añadido al entorno la capacidad de modularizar gramáticas dividiendo las mismas en fragmentos más simples, cada uno de los cuáles puede situarse en un archivo diferente. Con el fin de controlar posibles colisiones de nombres, se ha introducido también un mecanismo de espacio de nombres. Así

mismo, como siguiente paso se ha permitido descomponer gramáticas en base a *aspectos semánticos*, cada uno de las cuáles define un determinado aspecto del procesamiento. Por último, se ha permitido que los aspectos asociados con una misma tarea difieran en sus gramáticas incontextuales subyacentes, con el fin de flexibilizar dicho mecanismo de modularización. Los contenidos de esta sección están basados en [Temprado et al. 2010b].

3.4.1 Mecanismos de modularización básicos

El mecanismo de modularización empleado en la nueva versión de XLOP se basa en un sistema de espacios de nombres muy similar al que posee Java. El espacio de nombres permite desambiguar los distintos elementos de una gramática pertenecientes a contextos diferentes. De esta manera, cada fragmento que forma parte de una especificación más compleja se puede especificar en un archivo diferente, permitiendo la reutilización de estos fragmentos en otras especificaciones y facilitando su mantenimiento. Nótese que, de esta forma, las gramáticas XLOP pueden estar formadas, en la nueva versión, por la integración de un conjunto de fragmentos separados.

Conceptualmente, los símbolos no terminales utilizados en las nuevas especificaciones XLOP pertenecen a *espacios de nombres*:

- Para indicar el espacio de nombres de los símbolos no terminales de un fragmento, en la cabecera del archivo de especificación se indica la palabra "namespace", seguida de dicho espacio de nombres. Por defecto, en caso de no declarar el espacio de nombres, cada no terminal del fragmento corresponde al espacio de nombres por defecto "default".

- Para indicar el espacio de nombres al que pertenece un no terminal particular, debe usarse una orden "qualify". La cualificación se realiza especificando: "qualify", seguido del nombre del no terminal, seguido de "as", seguido del espacio de nombres del nuevo no terminal, seguido de ".", y seguido del nombre de dicho no terminal. Es posible especificar cero o más cualificaciones, pero no es posible cualificar un mismo no terminal de dos maneras diferentes.

- Por último, es posible cualificar explícitamente cada ocurrencia de no terminal con un espacio de nombres, seguido de ".", seguido del nombre del no terminal.

De esta forma, es importante entender el mecanismo de espacio de nombres en XLOP como un mero mecanismo de cualificación, sin ningún otro tipo de connotación semántica (p.ej. *importación* de símbolos).

```
namespace spaceOne

A ::= <tag> B T </tag> {
   h of B = id of <tag>
   s of A = s of B
}

T ::= <info/> {
   s of T = concatena(h of T,
            data of <info>)
}
```

Fragmento 1

```
namespace spaceOne
qualify T as spaceTwo.T


B ::= T {
   h of T = h of B
   s of B = s of T
}
```

Fragmento 2

```
namespace spaceTwo

D ::= T {
   h of T = h of D
   s of D = s of T
}

T ::= #pcdata {
   s of T = concatena(h of T,
            text of #pcdata)
}
```

Fragmento 3

```
spaceOne.A ::= <tag> spaceOne.B
               spaceOne.T </tag> {
   h of spaceOne.B = id of <tag>
   s of spaceOne.A = s of spaceOne.B
}

spaceOne.T ::= <info/> {
   s of spaceOne.T = concatena(h of
        spaceOne.T, data of <info>)
}

spaceOne.B ::= spaceTwo.T {
   h of spaceTwo.T = h of spaceOne.B
   s of spaceOne.B = s of spaceTwo.T
}

spaceTwo.T ::= #pcdata {
   s of spaceTwo.T = concatena(h of
        spaceTwo.T, text of #pcdata)
}
```

Gramática Resultante

Figura 3.4.1. Ilustración de los mecanismos de modularidad en XLOP.

A fin de ejemplificar el mecanismo, considérese la Figura 3.4.1. Todos los no terminales del fragmento 1 están en el espacio de nombres *spaceOne*. Dado que el fragmento 2 define el no terminal B en el fragmento 1 con una nueva producción, fija *spaceOne* como el espacio de nombres por defecto para sus símbolos. Así mismo, dado que, en el cuerpo de dicha producción, desea utilizarse el no terminal T que está en otro espacio de nombres (*spaceTwo*), se especifica dicho hecho con una orden “qualify”. Por último, el fragmento 3 introduce una regla para dicho no terminal T, así como una nueva regla para un nuevo terminal D, que también residirá en el espacio de nombres *spaceTwo*. La reunión de los tres fragmentos es

equivalente a la gramática resultante mostrada en la Figura 3.4.1. Obsérvese que, en dicha gramática, la regla D ::= T de Gramática C ha desaparecido. Esto es debido a que, en el proceso de combinación de fragmentos, se detectan los símbolos inaccesibles (en este caso, D), y se eliminan las producciones asociadas a los mismos. Efectivamente, la implementación del sistema de módulos posee un algoritmo de limpieza de gramáticas que elimina aquellas reglas que no se utilizarán en el procesamiento. Para que dicho algoritmo funcione correctamente, es necesario señalar, en la interfaz de XLOP, qué fragmento posee el axioma.

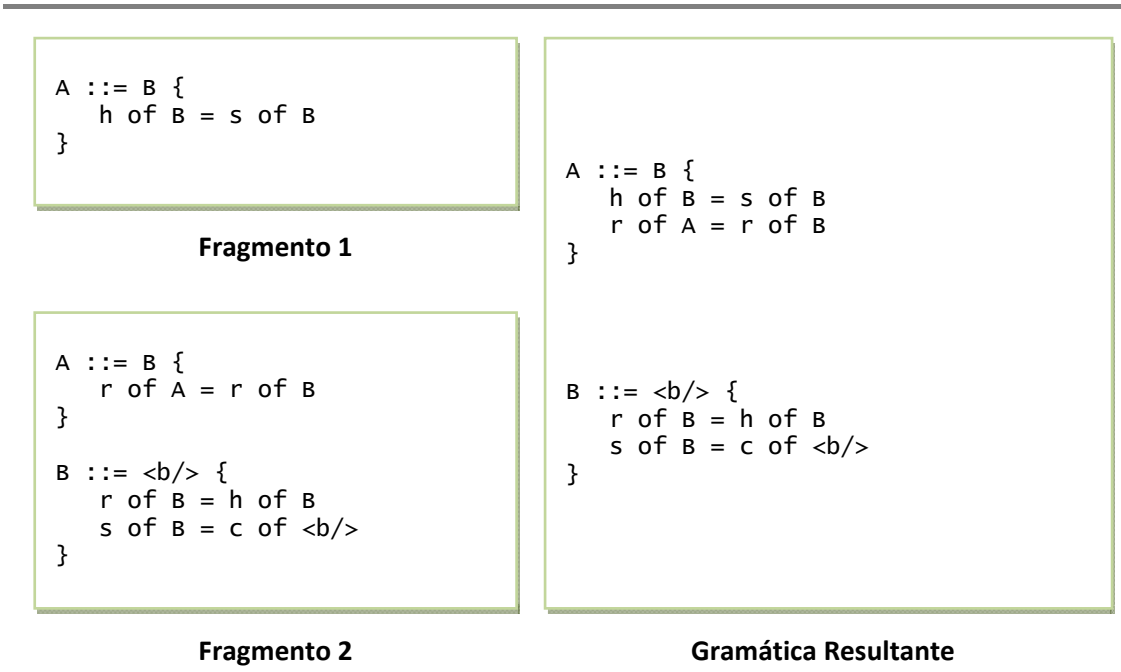


Figura 3.4.2. Ejemplo de fusión de producciones.

3.4.2 Aspectos semánticos

El siguiente paso en la introducción en XLOP de mecanismos de modularidad es el permitir que los fragmentos representen únicamente gramáticas de atributos parciales, en las que las reglas no tienen porque definir ecuaciones para todos los atributos que deben ser definidos en el contexto de las mismas. Para ello, XLOP fusiona aquellas reglas que tienen parte sintáctica común. El resultado es, como ya se ha indicado, una estrategia de modularización basada en aspectos semánticos, similar a la propuesta en [Kastens & Waite 1994].

La Figura 3.4.2 muestra un ejemplo. Obsérvese cómo, en la gramática resultante, prevalece una única regla con parte sintáctica A ::= B, que amalgama tanto las ecuaciones en el fragmento 1 como las ecuaciones en el fragmento 2. Esta facilidad es esencial, como ya se ha indicado, para implantar una estrategia de modularización por aspectos semánticos.

3.4.3 Especificaciones basadas en múltiples vistas sintácticas

Como hemos visto en la sección anterior, la especificación modular nos permite descomponer procesamientos complejos en aspectos más simples, y especificar éstos con fragmentos XLOP. Aunque la especificación modular en XLOP permite realizar dicha descomposición, es posible que distintos aspectos de procesamiento puedan ser realizados más adecuadamente sobre estructuras sintácticas diferentes. Estas gramáticas serán equivalentes entre sí, en el sentido de generar el mismo lenguaje, pero diferirán en el tipo de árboles sintácticos que asocian a los documentos.

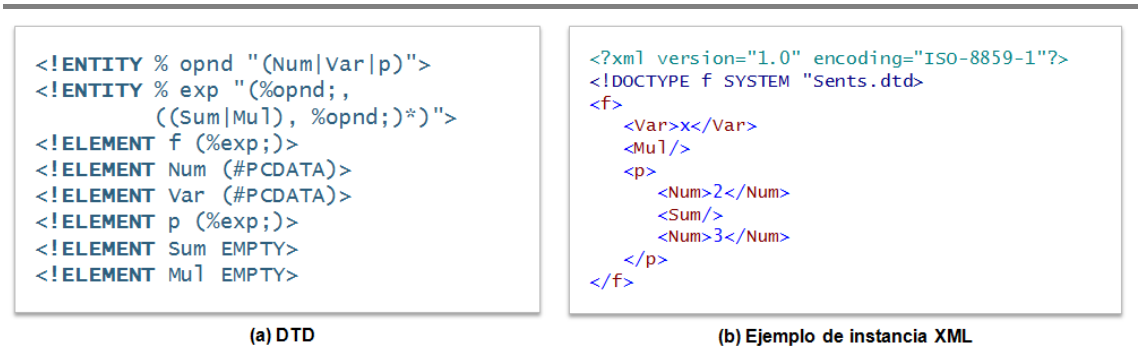


Figura 3.4.3. DTD e instancia XML para las fórmulas aritméticas.

En esta sección se introduce cómo utilizar múltiples vistas sintácticas en una especificación XLOP. Para ilustrar los conceptos, se considerará el ejemplo mostrado en la Figura 3.4.3, que presenta una DTD para codificar fórmulas aritméticas más sofisticadas que las presentadas en el apartado 3.2.1, junto a una instancia XML que codifica la fórmula "x*(2+3)". En este ejemplo, una fórmula, delimitado por las etiquetas *f*, puede constar a su vez de subfórmulas delimitadas por las etiquetas *p*, equivalente a los paréntesis. Los operadores de suma (*Sum*) y multiplicación (*Mul*) se representan como etiquetas vacías. Las variables quedan especificadas entre las etiquetas *Var*, y los números entre las etiquetas *Num*.

Una *vista sintáctica* es una caracterización, como una gramática independiente de contexto, de un lenguaje XML. De esta forma, si una especificación introduce vistas sintácticas alternativas, es necesario asegurar que dichas vistas son formalmente equivalentes, en el sentido de generar el mismo lenguaje, aunque, por supuesto, difieran en cómo estructuran las diferentes secuencias de los distintos elementos XML y fragmentos de texto en los contenidos de otros elementos. Igualmente, cada vista debe ser equivalente a la caracterización, como una gramática documental, del lenguaje en sí. Esta definición de vista, sin embargo, reviste un problema: ¿cómo se puede comprobar de forma efectiva (es decir, mediante un algoritmo) si dicha equivalencia se cumple? Para abordar este problema, en principio irresoluble [Hopcroft 1979], se imponen varias restricciones sobre las vistas, y se adopta el siguiente proceso (actualmente en fase de implementación en XLOP):

- Primeramente, se realiza la traducción de la gramática del documento a una gramática EBNF equivalente. A esta caracterización del lenguaje de marcado la denominaremos *núcleo sintáctico* del lenguaje, y a sus símbolos, *símbolos del núcleo*. Este proceso, se puede realizar de manera sistemática basándose en la definición de los distintos tipos de elementos XML, de tal manera que, a cada tipo de elemento le corresponda un símbolo no terminal, y a éste una regla EBNF en cuya parte derecha se describe, mediante una expresión regular, los posibles contenidos de dicho tipo de elemento.

- Por su parte, cada vista sintáctica debe describir la estructura de cada símbolo no terminal del núcleo como una gramática *no-autoembebible* [Nederhof 2000]. Una gramática *no-autoembebible* es una gramática incontextual que no admite derivaciones de tipo $A=>^{*}\alpha A\beta$, siendo α y β cadenas no vacías. Debido a que este tipo de gramáticas generan lenguajes regulares [Nederhof 2000], es posible utilizar algoritmos estándares para comprobar la equivalencia entre dichas gramáticas no-autoembebibles con cualquier otro descriptor que genere un lenguaje regular (p.ej. una expresión regular, un autómata finito, u otra gramática autoembebible). Así mismo, dado que las gramáticas no-autoembebibles no imponen ningún tipo de restricción en la forma de las reglas sintácticas, con la excepción de la condición indicada anteriormente, desde un punto de vista pragmático representan mecanismos muy convenientes a la hora de describir gramaticalmente lenguajes regulares (como pueden ser, por ejemplo, los generados por modelos de contenido XML).

- Por último, cada gramática no-autoembebible debe ser equivalente a la expresión regular que define el no terminal del núcleo asociado.

Todas las restricciones impuestas por este proceso pueden comprobarse automáticamente. Así mismo, si se consideran modelos de contenidos *no ambiguos*, y se obliga a que cada vista sintáctica sea determinista (p.ej. LALR(1)), es posible verificarlas de manera eficiente [Steams & Hunts 1985].

<table>
<tr><th>Núcleo sintáctico</th><th>Vista sintáctica 1</th><th>Vista sintáctica 2</th></tr>
<tr><td rowspan="7">f → <f> (n|v|p) ((a|m) (n|v|p))* </f></td><td colspan="2">f → <f> exp </f></td></tr>
<tr><td>exp → opnd rexp</td><td>exp → expr</td></tr>
<tr><td>rexp → op opnd rexp</td><td>expr → expr a term</td></tr>
<tr><td>rexp → λ</td><td>expr → term</td></tr>
<tr><td>op → a</td><td>term → term m opnd</td></tr>
<tr><td>op → m</td><td>term → opnd</td></tr>
<tr><td colspan="2">opnd → n
opnd → v
opnd → p</td></tr>
<tr><td>n → <Num> #pcdata </Num></td><td colspan="2">n → <Num> #pcdata </Num></td></tr>
<tr><td>v → <Var> #pcdata </Var></td><td colspan="2">v → <Var> #pcdata </Var></td></tr>
<tr><td>p → (n|v|p) ((a|m) (n|v|p))* </td><td colspan="2">p → exp </td></tr>
<tr><td>a → <Sum/></td><td colspan="2">a → <Sum/></td></tr>
<tr><td>m → <Mul/></td><td colspan="2">m → <Mul/></td></tr>
</table>

Figura 3.4.4. Núcleo sintáctico y vistas sintácticas equivalentes.

La Figura 3.4.4 muestra el núcleo sintáctico y dos vistas sintácticas equivalentes. En cada vista sintáctica, la definición de cada símbolo no terminal del núcleo se proporciona mediante una gramática no-autoembebible, y puede probarse equivalente con la expresión regular de la parte derecha de la correspondiente regla EBNF obtenida. Estas restricciones pueden comprobarse, como se ha indicado, de manera automática. En concreto, las vistas de la Figura 3.4.4 cumplen con las premisas descritas, y, tanto, también serán equivalentes entre sí. Estas vistas sintácticas han surgido por la necesidad de disponer de varias vistas especializadas en un aspecto concreto del procesamiento. La vista sintáctica 1 resulta más adecuada para procesamientos que no necesitan tratar explícitamente con la precedencia o asociatividad de los operadores. Por su parte, la vista sintáctica 2 es más adecuada para procesamientos que sí necesitan una clara distinción de precedencia y asociatividad entre operadores.

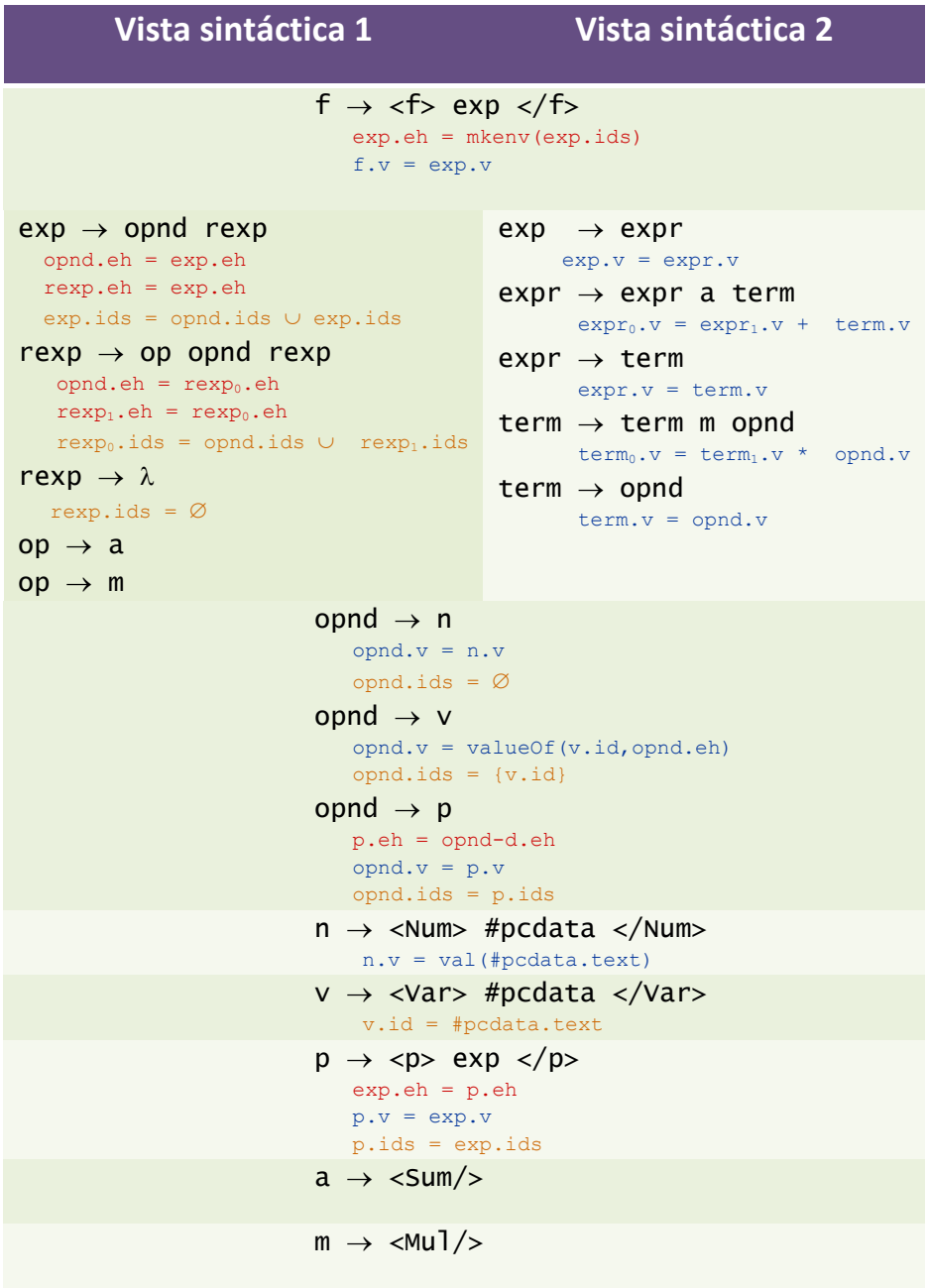


Figura 3.4.5. Gramática de atributos multivista. Propagación del entorno (en rojo), colección de variables (en naranja) y cálculo del valor (en azul).

Las *gramáticas de atributos multivista* son gramáticas de atributos especificadas a partir de distintos fragmentos formulados sobre varias vistas sintácticas del mismo lenguaje. La Figura 3.4.5 muestra la caracterización del procesamiento orientado a evaluar las expresiones marcadas mediante una gramática de atributos basada en múltiples vistas sintácticas. En concreto, el procesamiento se ha descompuesto en tres aspectos: una colección de variables (en naranja), una propagación del entorno (en rojo) y el cálculo del valor (en azul). Como puede observarse en la Figura 3.4.5 y como ya hemos mencionado anteriormente, para el cálculo del valor es conveniente contar con una gramática que refleje la precedencia y

asociatividad entre operadores de manera explícita, o lo que es lo mismo, utilizar la vista sintáctica 2. Para las demás tareas, es suficiente con utilizar una vista estructuralmente más sencilla: la vista sintáctica 1.

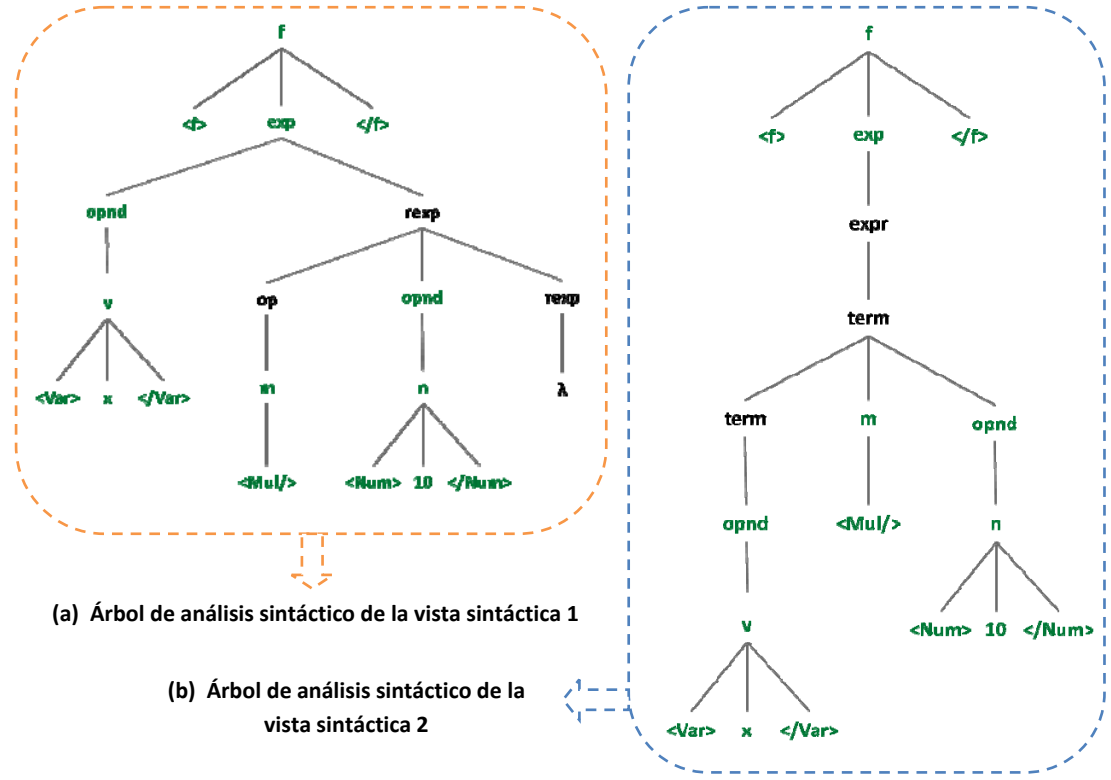


Figura 3.4.6. Árboles de análisis sintáctico para cada vista sintáctica de la Figura 3.4.4.

La Figura 3.4.6 muestra el árbol de análisis sintáctico correspondiente a cada vista sintáctica de la Figura 3.4.4. En concreto, la Figura 3.4.6a es el árbol de análisis sintáctico generado por el análisis de la sentencia "x*10" con la vista sintáctica 1; y la Figura 3.4.6b es el árbol de análisis sintáctico generado por el análisis de esa misma sentencia, pero con la vista sintáctica 2. Ambas representaciones pueden realizarse de manera compacta en un único bosque utilizando un Shared-Packed Parse Forest (SPPF, véase la sección 2.3.6). La Figura 3.4.7a muestra el SPPF resultante de la unión de los árboles de la Figura 3.4.6. En verde se han señalado aquellos subárboles que quedan compartidos, correspondientes a la misma estructura que comparten ambas vistas. Sin embargo, el nodo *exp* (en rojo) ha sido empaquetado debido a una ambigüedad en la estructura correspondiente a cada vista.

En el contexto de las gramáticas de atributos multivista, el SPPF no se utiliza simplemente como una manera de evitar el crecimiento exponencial de los árboles de análisis sintáctico, como ocurre con la aplicación de los métodos GLR al procesamiento del lenguaje natural, sino como una manera de integrar la evaluación de los atributos asignados a los nodos de cada árbol correspondiente a cada aspecto semántico. Este hecho se ejemplifica en la Figura 3.4.7b,

donde se muestra el flujo de información durante el proceso de evaluación inducido por la gramática basada en múltiples vistas sintácticas de la Figura 3.4.5 en el SPPF de la Figura 3.4.7a. Este proceso de evaluación se especificará en el próximo capítulo.

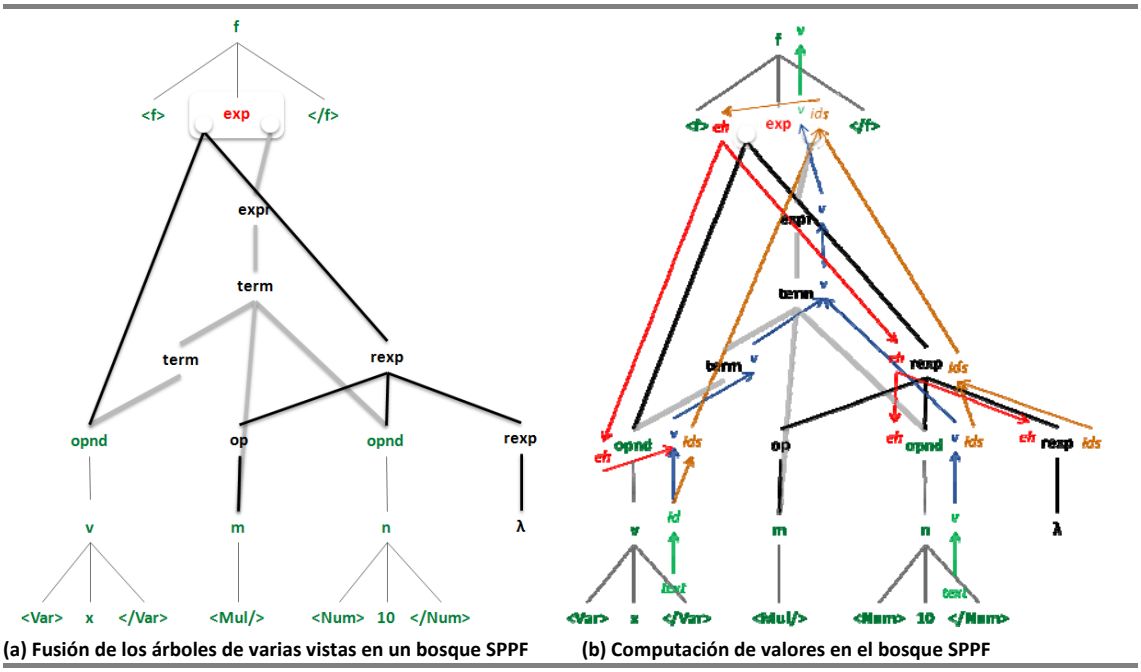


Figura 3.4.7. Bosque SPPF correspondiente a la combinación de los árboles de Figura 3.4.6 y computación de valores sobre el SPPF.

Nótese que, en el lenguaje básico de especificación de XLOP, no es posible especificar directamente gramáticas multivista. Esto se debe a que, en la gramática resultante, en todas las reglas será necesario definir todos los atributos sintetizados de la cabeza, así como todos los atributos heredados del cuerpo. Sin embargo, en una gramática basada en múltiples vistas sintácticas, esto no es necesariamente cierto para los símbolos del núcleo. Efectivamente, dado un símbolo no terminal del núcleo *A*, basta que:

- Sus atributos sintetizados se definan en *una cualesquiera* de las vistas.
- Fijada una vista, se defina el mismo conjunto de atributos heredados en cada ocurrencia de *A*, y que dichos atributos se definan únicamente en dicha vista.

Para permitir expresar este hecho, el lenguaje de especificación de XLOP se extiende para permitir definir diferentes vistas sintácticas. Para ello se utiliza la orden **views**. Dicha orden tiene como parámetros:

- Una secuencia de identificadores de vistas sintácticas.
- La secuencia de símbolos del núcleo de dichas vistas.

```
namespace expresiones
view idsRecollection

exp ::= opnd rexp
{ ids of exp = union(ids of opnd,
ids of rexp) }
rexp ::= op opnd rexp
{ ids of rexp(0) =
 union(ids of opnd, ids of rexp(1))}
rexp ::=
{ ids of rexp = conjuntoVacio() }
op ::= a {}
op ::= m {}
a ::= <Sum/> {}
m ::= <Mul/> {}
opnd ::= n
{ ids of opnd = conjuntoVacio() }
opnd ::= v
{ ids of opnd = conjunto(id of v) }
opnd ::= p
{ ids of opnd = ids of p }
n ::= <Num> #pcdata </Num> {}
v ::= <Var> #pcdata </Var>
{ id of v = text of #pcdata }
p ::= <p> exp </p>
{ ids of p = ids of exp }
```

Fragmento 1

```
namespace expresiones
view environmentPropagation

f ::= <f> exp </f>
{ eh of exp = mkenv(ids of exp) }
exp ::= opnd rexp
{ eh of opnd = eh of exp
  eh of rexp = eh of exp }
rexp ::= op opnd rexp
{ eh of opnd = eh of rexp(0)
  eh of rexp(1) = eh of rexp(0) }
rexp ::= {}
op ::= a {}
op ::= m {}
a ::= <Sum/> {}
m ::= <Mul/> {}
opnd ::= n {}
opnd ::= v {}
opnd ::= p
{ eh of p = eh of opnd }
n ::= <Num> #pcdata </Num> {}
v ::= <Var> #pcdata </Var> {}
p ::= <p> exp </p>
{ eh of exp = eh of p }
```

Fragmento 2

```
namespace expresiones
view evaluation

f ::= <f> exp </f>
{ v of f = v of exp }
exp ::= expr
{ v of exp = v of expr }
expr ::= expr a term
{ v of expr(0) = suma(v of expr(1),
 v of term) }
expr ::= term
{ v of expr = v of term }
term ::= term m opnd
{ v of term(0) = multiplicacion(v of
term(1), v of opnd) }
term ::= opnd
{ v of term = v of opnd }
a ::= <Sum/> {}
m ::= <Mul/> {}
opnd ::= n
{ v of opnd = v of n }
opnd ::= v
{ v of opnd = valueOf(id of v,
 eh of opnd) }
opnd ::= p
{ v of opnd = v of p }
n ::= <Num> #pcdata </Num>
{ v of n = val(text of #pcdata) }
v ::= <Var> #pcdata </Var> {}
p ::= <p> exp </p>
{ v of p = v of exp }
```

Fragmento 3

```
namespace expresiones

views {idsRecollection
       environmentPropagation
       evaluation}

cores {f, n, v, p, a, m}
```

Declaración de las vistas

Figura 3.4.8. Especificación modular XLOP de la gramática de atributos multivista de la Figura 3.4.5.

Así mismo, cada fragmento de gramática de atributos puede comenzar mediante una orden **view**, que introduce la vista sintáctica en la que se incluye dicho fragmento. Esto permite relajar la restricción en el sentido indicado:

- En todos los fragmentos asociados con un conjunto de vistas sintácticas introducido mediante **views**, todas las reglas que tengan en su cabeza un símbolo del núcleo deberán definir, en su conjunto, una única vez cada atributo sintetizado de dicha cabeza.

- En todos los fragmentos asociados con una vista particular, para cada ocurrencia de un símbolo del núcleo se definen exactamente los mismos atributos heredados, y dichos atributos se definen únicamente en dichos fragmentos.

Esto permite comprobar en XLOP el resto de las restricciones asociadas con las gramáticas basadas en múltiples vistas sintácticas (en particular, la equivalencia de las vistas, aún en ausencia explícita del núcleo sintáctico).

La Figura 3.4.8 muestra la especificación modular XLOP de la gramática de atributos multivista de la Figura 3.4.5. En dicha especificación, el fragmento 1 (formulado sobre la vista 1) caracteriza la recolección de variables, el fragmento 2 (también formulado sobre la vista 1) caracteriza la propagación del entorno, y el fragmento 3 (formulado, esta vez, sobre la vista 2) caracteriza el proceso de cálculo del valor de la expresión. Por último, el fragmento 4 se utiliza para declarar el conjunto de vistas.

3.4.4 Extensión de la sintaxis

SpecXLOP ::= *Cabecera* {*ElementoCuerpo*}+

Cabecera ::= (**namespace** *EspacioDeNombres*)? {**qualify** *NoTerminal* **as** *NoTerminal*}* (**view** *NombreVista*)?

ElementoCuerpo ::= *Vistas* | *Regla*

Vistas ::= **views** '{' {*nombreVista*}+ '}' **cores** '{' {*NoTerminal*}+ '}'

Regla ::= *NoTerminal* '::=' { *ElementoSintactico* }* '{' { *Ecuacion* }* '}'

ElementoSintactico ::= *NoTerminal* | **#pcdata** | *Elemento*XML

*Elemento*XML ::= *EtiquetaApertura* { *ElementoSintactico* }* *EtiquetaCierre* |
EtiquetaElmVacio

Ecuacion ::= *ReferenciaAtributo* '=' *ExpresionSemantica*

ExpresionSemantica ::= *Funcion* '(' (*ExpresionSemantica* { , *ExpresionSemantica* }*)? ')' |
ReferenciaAtributo | *ValorLiteral*

ReferenciaAtributo ::= *Atributo* **of** (*NoTerminal* | **#pcdata** | *EtiquetaApertura*)
('(' *NumeroOcurrencia* ')')?

Figura 3.4.9. Extensión de la sintaxis.

La Figura 3.4.9 resume la sintaxis del lenguaje de especificación de XLOP que resulta de incluir las extensiones descritas en esta sección. En esta sintaxis, *NoTerminal* representa un nombre de no terminal posiblemente cualificado con un espacio de nombres.

3.5 A modo de conclusión

Este capítulo ha revisado la versión inicial de XLOP, el sistema de procesamiento de documentos XML basado en gramáticas de atributos desarrollada en el 2009 [Martínez & Temprado 2009] y [Sarasa et al. 2009d], y se ha introducido los aspectos conceptuales de la propuesta realizada en este trabajo de fin de máster. En concreto:

- Se ha presentado el enfoque gramatical de XLOP. La estructura de los documentos XML se puede describir mediante una gramática incontextual que permite generar un árbol de análisis más completo y adecuado a la descripción del procesamiento de dichos documentos que el árbol documental. La incorporación de atributos y ecuaciones semánticas a dicha gramática, nos permite describir el procesamiento de los documentos como una computación de valores de atributos semánticos asociados a los nodos del árbol de análisis sintáctico.

- Se ha detallado el lenguaje de especificación de la versión inicial de XLOP. El proceso de generación de aplicaciones de procesamiento XML con XLOP comienza con la especificación de una gramática de atributos: la gramática XLOP. Dicha gramática se debe definir mediante una notación especial y debe cumplir una serie de pautas y restricciones contextuales.

- Se ha detallado la metodología de desarrollo de aplicaciones de procesamiento XML con XLOP, y se ha analizado, mediante un caso de estudio, cómo es posible utilizar XLOP en el desarrollo de una aplicación no trivial.

- Se ha introducido el sistema de módulos que utilizará la nueva versión de XLOP. Con este sistema, XLOP será capaz de modularizar especificaciones complejas en términos de fragmentos más simples.

- Por último, se han introducido las especificaciones basadas en múltiples vistas sintácticas. La descomposición en aspectos puede originar aspectos que discrepen en las estructuras sintácticas más convenientes para su realización. Las especificaciones basadas en múltiples vistas sintácticas permiten armonizar, en una misma especificación, la citada heterogeneidad sintáctica, admitiendo simultáneamente distintas gramáticas incontextuales durante el mismo procesamiento.

El siguiente capítulo describe cómo soportar, en tiempo de ejecución, las gramáticas de atributos basadas en múltiples vistas sintácticas. Para ello, se adoptará el algoritmo GLR como

núcleo básico de análisis sintáctico, junto con un nuevo mecanismo de evaluación en XLOP orientado a los SPPFs generados por dicho algoritmo.

Capítulo 4
Modelo de Evaluación para Gramáticas de Atributos Multivista: Implementación en XLOP

4.1 Introducción

A fin de validar la factibilidad de la propuesta de extensión de XLOP realizada en el capítulo anterior, se ha llevado a cabo el desarrollo e implementación de varios de los aspectos descritos en dicha propuesta. Más concretamente:

- Se ha incluido en el entorno el soporte básico para especificaciones modulares descrito en la sección 3.4.1 del capítulo anterior: espacios de nombres, y el mecanismo de modularización basado en la separación de aspectos semánticos, que permite fusionar las reglas que comparten la misma parte sintáctica.

- Se ha añadido a XLOP un nuevo algoritmo de análisis sintáctico basado en el algoritmo GLR (véase la sección 2.3.4 del Capítulo 2). Dicho algoritmo es la contrapartida, en la extensión realizada, al algoritmo LALR(1) soportado por CUP en la versión 1.0 del entorno.

- Se ha desarrollado un nuevo modelo de evaluación de atributos dirigido por *demanda*, que supone la contrapartida, en el nuevo modelo de ejecución, al modelo dirigido por el flujo de datos presentado en la sección 3.2.3 del capítulo anterior.

Dicho desarrollo ha obviado la parte relativa a la comprobación de las restricciones contextuales adicionales asociadas con las gramáticas multivista (véase la sección 3.4.3 del capítulo anterior), aspectos en los que se está trabajando actualmente. No obstante, el desarrollo llevado a cabo sí se estima suficiente para chequear la factibilidad práctica de la propuesta realizada. En dicho desarrollo, mientras que la implementación concreta del algoritmo GLR es directa a partir del estudio realizado en la sección 2.3, y la incorporación del soporte de especificaciones modulares es sencilla a partir del trabajo previo [Martínez & Temprado 2009], el diseño e implementación del modelo de evaluación para gramáticas de atributos multivista y su acoplamiento con el algoritmo GLR sí implica una especial complejidad. En este capítulo se detalla dicho modelo de evaluación y su implementación concreta en el entorno XLOP.

4.2 El modelo de evaluación para gramáticas de atributos multivista

4.2.1 Visión general

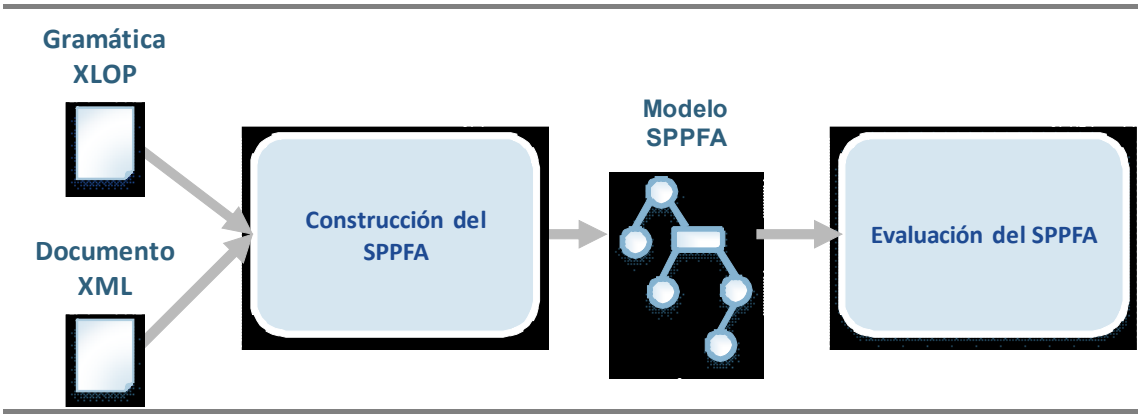


Figura 4.2.1. Visión general del modelo de evaluación para gramáticas de atributos multivista.

La Figura 4.2.1 muestra el esquema general del modelo de evaluación para gramáticas de atributos multivista:

- La primera etapa en este modelo es la construcción del *bosque de análisis sintáctico atribuido* asociado al documento: el SPPFA. Dicha construcción está guiada por el algoritmo de análisis GRL. Más concretamente, el proceso parte de una gramática de atributos multivista o gramática XLOP, y del documento XML a analizar. Aparte de representar la estructura del documento, tal y como haría un SPPF convencional construido por el algoritmo GLR, el modelo SPPFA está adaptado para almacenar el valor de los distintos atributos semánticos asociados a los nodos del bosque, y para albergar la forma de acceder y computar dichos valores en el bosque. Su estructura se detalla en la sección 4.2.2. La sección 4.2.3, por su parte, detalla dicha construcción.

- La segunda etapa se ocupa de la *evaluación* del SPPFA. La evaluación del modelo SPPFA se realiza siguiendo un mecanismo de evaluación por demanda, según el cual los diferentes valores de los atributos, asociados a los nodos del bosque, se computan únicamente en el momento en que son requeridos, siempre y cuando su valor no haya sido computado previamente. La estrategia es análoga a la utilizada en patrones para la evaluación de atributos en árboles atribuidos descritos en [Magnusson & Hedin 2007]. En esta fase, pues, se añade al modelo SPPFA la semántica computacional que especifica cómo obtener los distintos valores alojados en los distintos nodos del bosque y su manera de computarlos. La ejecución de la evaluación en sí, se reduce a la consulta de los valores de los atributos sintetizados del nodo raíz del SPPFA, y el mecanismo de evaluación por demanda se encarga del resto. La sección 4.2.4 detalla esta última fase. Dicho

mecanismo de evaluación radica, en última instancia, en la correcta ejecución de las ecuaciones de la gramática. Dicha ejecución se lleva a cabo a través de una serie de *procedimientos de cómputo*, uno para cada ecuación, que se generan a partir de la gramática. La sección 4.2.5 analiza, por último, dicho proceso de generación.

4.2.2 El modelo SPPFA

En la sección 2.3.8 del Capítulo 2 se mostró cómo el algoritmo GLR es capaz de producir un bosque que compacta los diversos árboles de análisis sintácticos de una frase en una única representación eficiente: el SPPF (sección 2.3.6). A partir del examen de la implementación del algoritmo GLR, es posible discernir dos tipos de nodos en el SPPF:

- Nodos SPPF terminales. Estos nodos se corresponden con las hojas del bosque.
- Nodos SPPF no terminales. Estos nodos se corresponden con un nodo del bosque asociado a un no terminal. Dicho nodo puede ser empaquetado o no. De hecho, los nodos empaquetados son nodos no terminales que tienen asociados más de una lista de hijos. Por tanto, a nivel de representación, no es necesario realizar una distinción de dichos nodos en el modelo de ejecución. Por otra parte, un nodo SPPF contiene, por defecto, el símbolo que representa, símbolo que tampoco es necesario tener en cuenta en nuestro modelo.

De esta forma, el modelo SPPFA surge de ampliar el modelo SPPF con campos para alojar atributos semánticos, así como con mecanismos para computar y acceder a sus valores. Más concretamente, los nodos SPPFA, al igual que los nodos SPPF, son de dos tipos:

- Nodos SPPFA no terminales. Además de los campos de un nodo SPPF no terminal, disponen de una lista para los atributos heredados, y otra lista para los atributos sintetizados. El campo contenedor del símbolo se reemplaza por una *lista de reglas*, que determina la regla sintáctica utilizada para añadir cada lista de hijos del conjunto de listas de hijos del nodo. Nótese que el valor de cada atributo perteneciente a un no terminal debe ser calculado con la ecuación semántica que lo computa. Debido a esto, dicho valor puede no haber sido calculado en el momento que se solicite. Es por ello que cada atributo tiene asociado un *gestor del valor*, un objeto que almacena el valor del atributo y, en caso de no haber sido calculado el valor cuando se realice la petición, que invoca a un *procedimiento de cómputo* para obtenerlo, almacenándolo para devolverlo directamente en invocaciones futuras. Para cada atributo el gestor correspondiente debe almacenar también una referencia al nodo de aquél en el que reside, que se corresponde con la cabeza de la producción en la que reside la ecuación que indica cómo computar dicho atributo. Concretamente, para los atributos sintetizados debe almacenar la referencia al nodo actual, mientras que para los atributos heredados debe almacenar la referencia al nodo padre. La Figura 4.2.2

muestra esquemáticamente la información almacenada en un nodo SPPF no terminal, comparándola con la almacenada en el nodo SPPF original. Nótese que el gestor de valor, aparte de contener el procedimiento de cómputo, el valor, y la referencia al nodo (contexto), contiene un *flag* que permite indicar si el valor ha sido o no ha sido ya computado.

- Nodos SPPFA terminales. Además de los campos de un nodo SPPF terminal, disponen de una lista de atributos léxicos. Nótese que, para estos atributos léxicos, el gestor del valor correspondiente puede restringirse a almacenar el valor del atributo, no precisándose procedimiento de cómputo ni información de control adicional alguna.

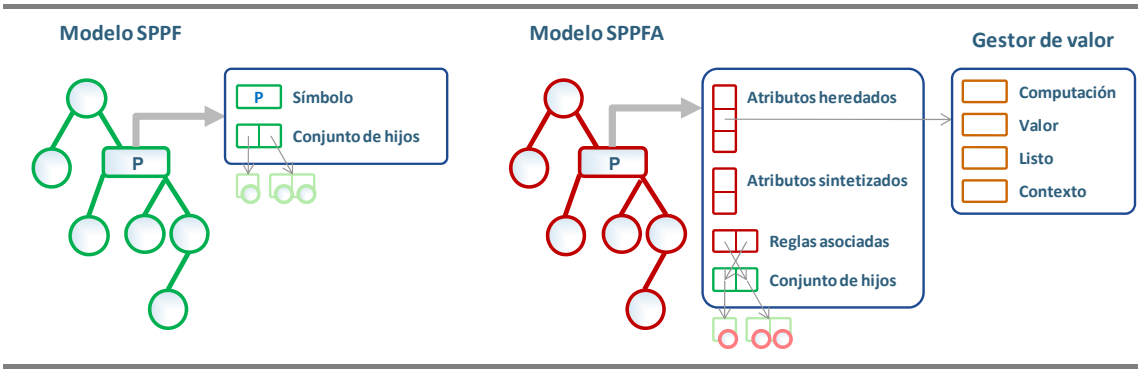


Figura 4.2.2. Estructura de nodos SPPF y nodos SPPFA no terminales.

4.2.3 Construcción del modelo SPPFA

La construcción del SPPFA se lleva a cabo mediante una ligera modificación del algoritmo GLR, a fin de que éste cree nodos SPPFA en lugar de nodos SPPF básicos. De esta forma:

- Cada vez que el algoritmo GLR realiza una acción de desplazamiento, se crea el correspondiente nodo SPPFA terminal. Dependiendo del tipo de token considerado, se almacenará en el mismo unos u otros atributos léxicos. De esta forma, para tokens de tipo #pcdata, se almacenará un atributo léxico `text`. Para tokens asociados con etiquetas de apertura, se almacenarán los correspondientes atributos XML asociados con las etiquetas. Para tokens asociados con etiquetas de cierre, no se almacenará atributo léxico alguno.

- Por su parte, cuando una reducción origina un nuevo nodo no terminal, se crea el SPPFA correspondiente, reservando espacio para sus atributos sintetizados y heredados, e inicializando el resto de sus estructuras internas. Seguidamente se añade la lista de hijos correspondiente, se asocia dicha lista de hijos con la regla

que ha originado la reducción, y se utilizan las ecuaciones de dicha regla para configurar de manera apropiada los gestores de valores, tanto del nuevo nodo creado a partir de las ecuaciones asociadas a sus atributos sintetizados, como de los nodos hijo a partir de las ecuaciones correspondientes a los atributos heredados de los mismos. El resto de las reducciones que conducen a dicho nodo implican únicamente esta segunda fase: registro de la lista de hijos, asociación de dicha lista con la producción correspondiente, y configuración de los gestores de valores. Nótese, así mismo, que, en base a las restricciones impuestas sobre las gramáticas multivista, es posible dimensionar exactamente el conjunto de listas de hijos de cada nodo.

La Figura 4.2.3 ejemplifica la construcción. La Figura 4.2.3a muestra un lenguaje XML muy simple, mientras que la Figura 4.2.3b muestra un documento de ejemplo. La Figura 4.2.3c muestra una gramática de atributos multivista definida sobre dos vistas sintácticas diferentes de dicho lenguaje. La Figura 4.2.3d muestra el SPPFA generado por el proceso de construcción dirigido por el algoritmo GLR a partir del documento de ejemplo. Obsérvese que hay un procedimiento de cómputo asociado a cada ecuación de la gramática, y que dichos procedimientos aceptan, como argumentos, una referencia al nodo del SPPFA que sirve como contexto para la evaluación de la ecuación. Dicho nodo será el asociado con la cabeza de la producción donde se definió la ecuación (el propio nodo en el que reside el atributo, para atributos sintetizados, o uno de los padres del nodo para los atributos heredados).

```
<!ELEMENT a (b)>
<!ELEMENT b (#PCDATA)>
<!ATTLIST b id ID #REQUIRED >
```

(a) DTD

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE a SYSTEM "Sents.dtd>
<a>
   <b id="x">x</b>
</a>
```

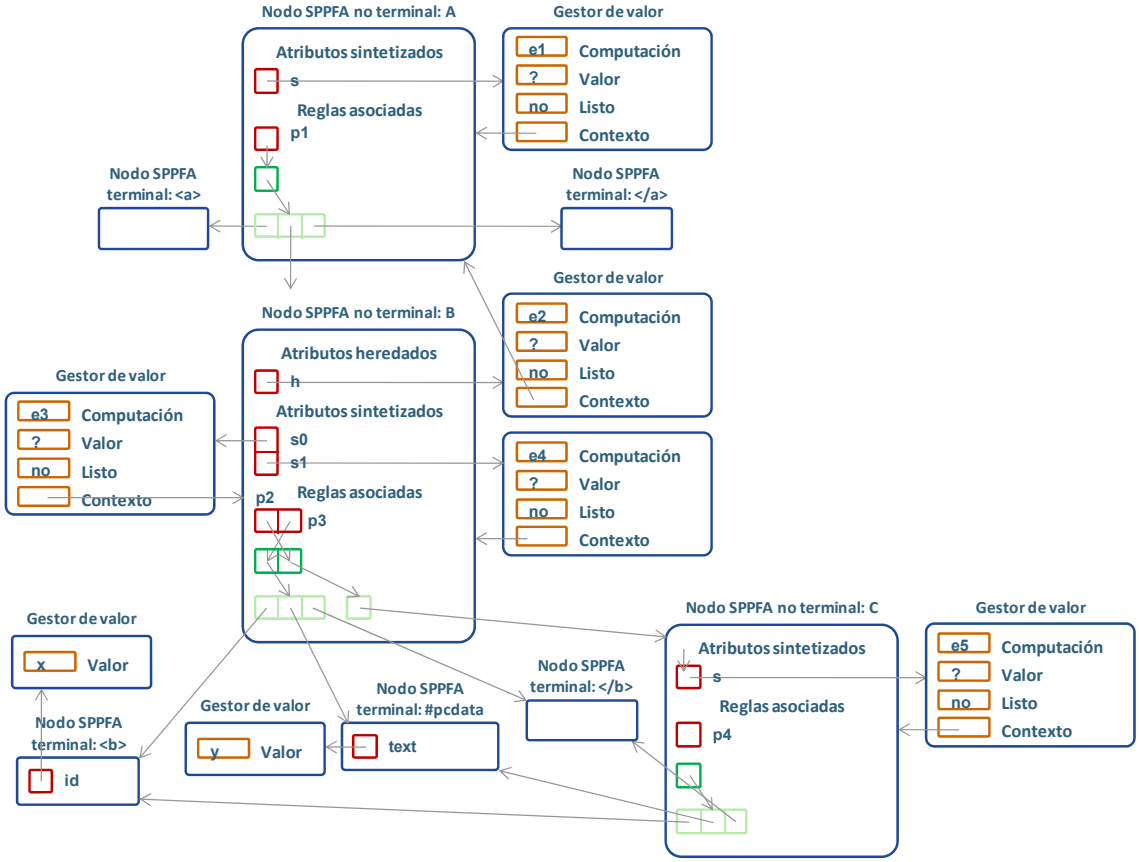
(b) Ejemplo de instancia XML

Vista sintáctica 1	Vista sintáctica 2
A → <a> B [p1] A.s = f(B.s1, B.s0) [e1] B.h = B.s0 [e2]	

B → C [p3]
B.s1 = g(C.s, B.h) [e4]
	B → #pcdata [p2]
	B.s0 = #pcdata.text [e3]

C → #pcdata [p4]
C.s = .id [e5]	

(c) Gramática multivista y nodos SPPFA asociados



(d) SPPFA

Figura 4.2.3. Ejemplo de SPPFA construido por el proceso de construcción basado en el algoritmo GLR.

4.2.4 Evaluación en el modelo SPPFA

El mecanismo de evaluación por demanda del modelo SPPFA reside en la manera de computar los valores de cada atributo asociado a los nodos del bosque. Concretamente, este cálculo se realiza a través del *gestor del valor* de un atributo. Más concretamente:

- Cuando se solicita el valor de un atributo, dicha solicitud se delega en el gestor del valor de dicho atributo.
- Si el gestor conoce el valor, lo devuelve directamente.
- En otro caso, el gestor *ejecuta* el procedimiento de cómputo asociado. Nótese que, tal y como se ejemplifica en la Figura 4.2.4, los procedimientos de cómputo *evalúan*, en realidad, la expresión funcional de la parte derecha de la ecuación, de acuerdo con la siguiente semántica:
  - La evaluación de una expresión del tipo $f(e_0, \ldots, e_k)$ supone evaluar en orden cada e_i, y seguidamente aplicar f sobre la lista de valores que resulta.
  - La evaluación de una referencia a un atributo supone solicitar el valor de dicho atributo. Dicho valor puede residir en el mismo nodo, en algún nodo padre, en algún nodo hijo, o en algún nodo hermano, pero, en cualquier caso, siempre es posible determinar la posición relativa al nodo de contexto (el nodo al que está asociado el gestor, para atributos sintetizados, o uno de los padres, para atributos heredados).

```
proc e1() {
  return f(ctx.getChild(p1,B).getSynAtt(s1).getValue(), ctx.getChild(p1,B).getSynAtt(s0).value())
}
proc e2 ()  {
  return ctx.getChild(p1,B).getSynAtt(s0).getValue()
}
proc e3 ()  {
  return ctx.getChild(p2,#pcdata).getLexicalAtt(text).value()
}
proc e4 ()  {
  return g(ctx.getChild(p3,C).getSynAtt(s).value(), ctx.getInhAtt(h).value())
}
proc e5()  {
  return ctx.getChild(p4,<b>).getLexicalAtt(id).value()
}
```

Figura 4.2.4. Procedimientos de cómputo asociados a los gestores de valores del SPPFA de la Figura 4.2.3d.

Una vez finalizado este proceso de evaluación, almacena el valor resultante, para servir futuras peticiones, y registra en el *flag* destinado a tal fin que dicho valor ha sido ya efectivamente computado.

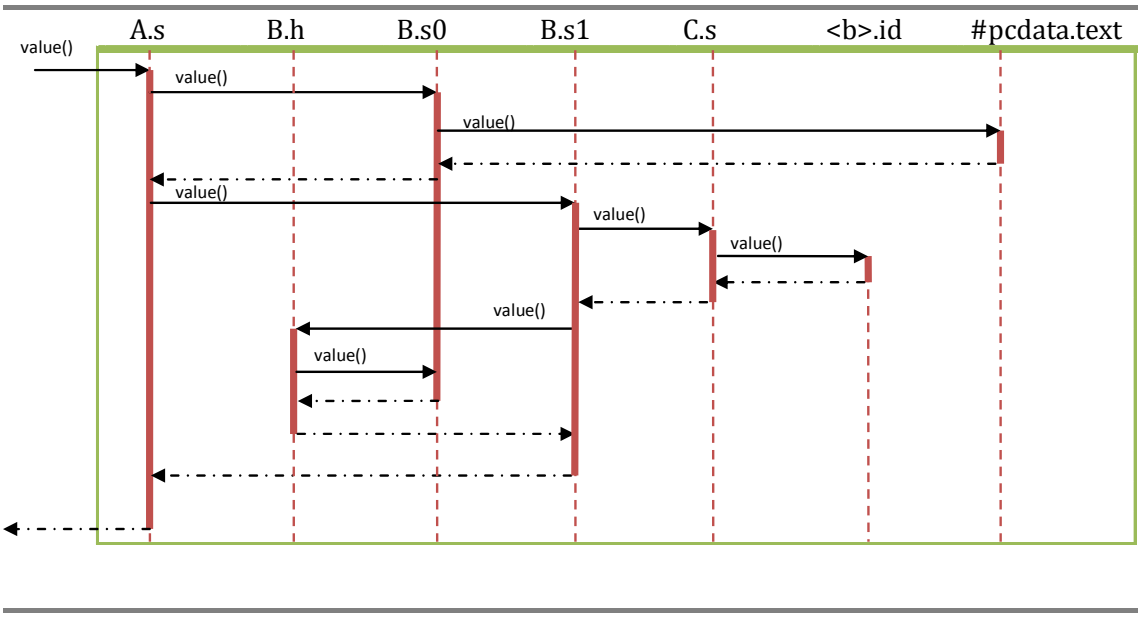


Figura 4.2.5. Proceso de evaluación del SPPFA de la Figura 4.2.3d.

El diagrama de secuencias de la Figura 4.2.5 ilustra el proceso de evaluación sobre el SPPFA de la Figura 4.2.3d. El proceso de evaluación se desencadena cuando se solicita el valor del atributo *s* de la raíz. Obsérvese que cuando, desde el procedimiento de cómputo para la ecuación *e2* se solicita el valor del atributo *s0* del nodo *B*, dicho valor se devuelve directamente al haber sido computado con anterioridad en el proceso de evaluación.

4.2.5 Generación de los procedimientos de cómputo

El modelo de evaluación depende, en última instancia, de los procedimientos de cómputo asociados a las ecuaciones. Dichos procedimientos pueden generarse automáticamente a partir de la gramática. Para ello, debe determinarse la forma de acceder a los atributos referidos en la expresión semántica de la ecuación. La forma de acceso obedece a las siguientes reglas:

- Si el atributo es un atributo heredado de la cabeza, puede encontrarse almacenado directamente en el nodo recibido como parámetro (véase, por ejemplo, el acceso a B.h en el procedimiento *e4* de la Figura 4.2.4).

- Si el atributo es un atributo sintetizado de un símbolo del cuerpo, el primer atributo referido estará almacenado en el correspondiente símbolo de la lista de hijos asociada a la producción en el nodo recibido como parámetro.

Es importante volver a resaltar que los nodos que los procedimientos de cómputo reciben los nodos asociados con la cabeza de la producción en la que se definen las correspondientes ecuaciones. En el caso de los atributos sintetizados, dicho nodo coincidirá con el nodo al que pertenecen dichos atributos, pero no así en el caso de los atributos heredados: en este caso, el nodo será alguno de los padres en el SPPFA. Es por ello que los gestores han de referir expresamente a dicho nodo.

4.3 Implementación en XLOP

4.3.1 El proceso de generación

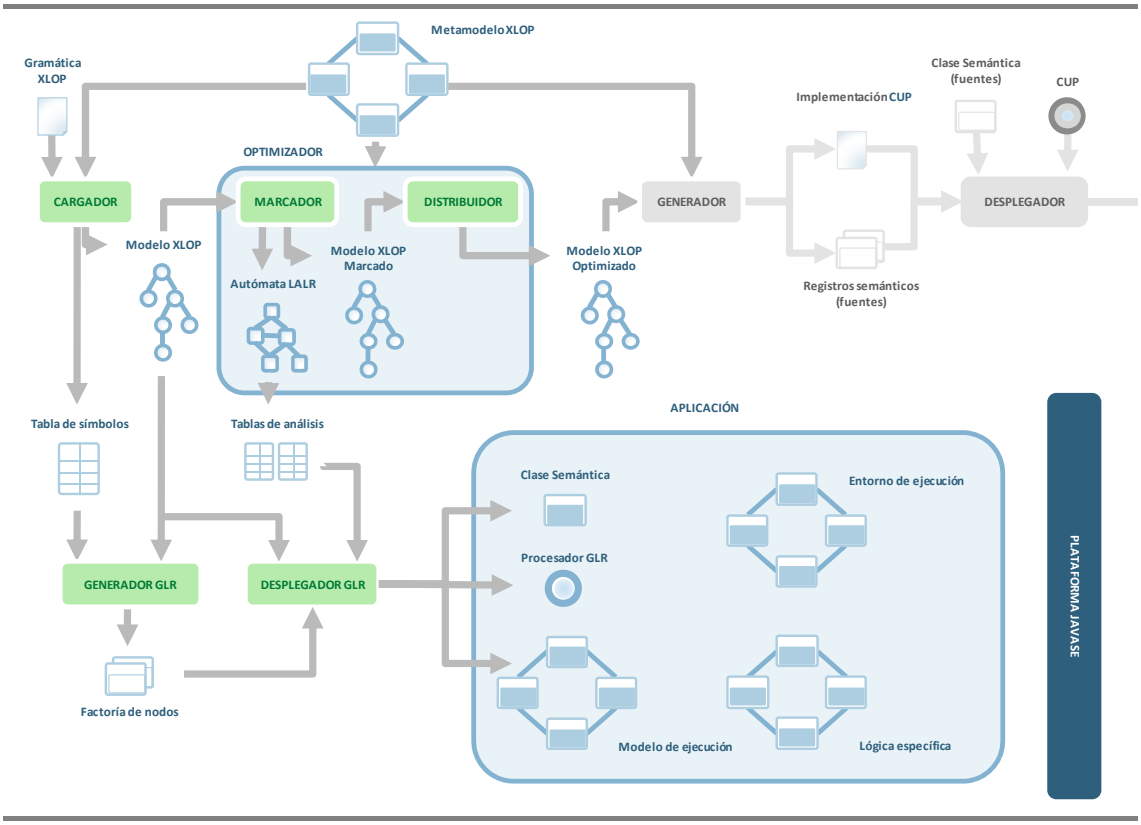


Figura 4.3.1. Nueva arquitectura de XLOP.

El proceso de generación en XLOP 1.0 (ver sección 3.2.3) se basaba en la generación de especificaciones (esquemas de traducción) para un generador de analizadores sintáctico (más concretamente, CUP) en las que se codificaba un mecanismo de evaluación dirigido por los datos. A partir de dichas especificaciones, el generador de CUP construía todos los elementos necesarios (autómata LALR(1), tablas de análisis, etc.) para la correcta ejecución de su algoritmo interno de tipo LR. Sin embargo, el modelo de ejecución para gramáticas de atributos multivista utiliza el algoritmo GLR. Por consiguiente, es necesario realizar importantes extensiones en la arquitectura de XLOP.

La Figura 4.3.1 muestra la nueva arquitectura de XLOP. En esta arquitectura:

- Se ha incorporado un nuevo módulo: el módulo *Generador GLR*, que proporciona soporte a las especificaciones modulares basadas en gramáticas de atributos multivista. Hay que resaltar que el antiguo módulo generador no se ha eliminado del sistema y se puede optar por usar un generador u otro según la opción deseada.

- El módulo *Cargador* ha sido modificado para que soporte especificaciones modulares.

- Por otra parte, la versión inicial de XLOP es capaz de construir el autómata LALR(1) característico de la gramática XLOP, a partir del modelo XLOP. Dicha construcción fue incluida en el módulo *Marcador*. Aprovechando dicho autómata LALR(1), en la extensión realizada se construyen las tablas de análisis que utilizará el algoritmo GLR.

- El módulo *Generador GLR* también produce la *Factoría de nodos* (sección 4.3.3) a partir del modelo XLOP y la tabla de símbolos que dan soporte al proceso de evaluación de atributos. Dichas clases contienen la traducción de las ecuaciones semánticas a operaciones Java, y permite vincular dichas ecuaciones con los respectivos atributos semánticos que computan, implementando los procedimientos de cómputo introducidos en el modelo de evaluación.

- El *Procesador GLR contiene* la implementación del procesador, cuyo núcleo se compone de la implementación del algoritmo GLR y del modelo de ejecución.

- El módulo *Desplegador GLR* proporciona al *Procesador GLR* todos los elementos que necesita para la creación del bosque de análisis sintáctico y su posterior evaluación, función realizada por las clases que componen el *Modelo de ejecución*.

- Los restantes *componentes* de la arquitectura inicial se conservan. El *Entorno de ejecución* se mantiene, y permite, de la misma manera, el análisis secuencial de los archivos XML a través de una implementación específica del parser SAX, que transmite por peticiones los elementos XML analizados al algoritmo GLR. Y de la misma manera que el módulo *Desplegador* original, el módulo *Desplegador GLR* agrupa todos los archivos necesarios, incluyendo la *clase semántica* y la *lógica especifica de la aplicación*, permitiendo generar un paquete con la implementación final de la aplicación de procesamiento XML lista para su uso.

4.3.2 Detalles de diseño

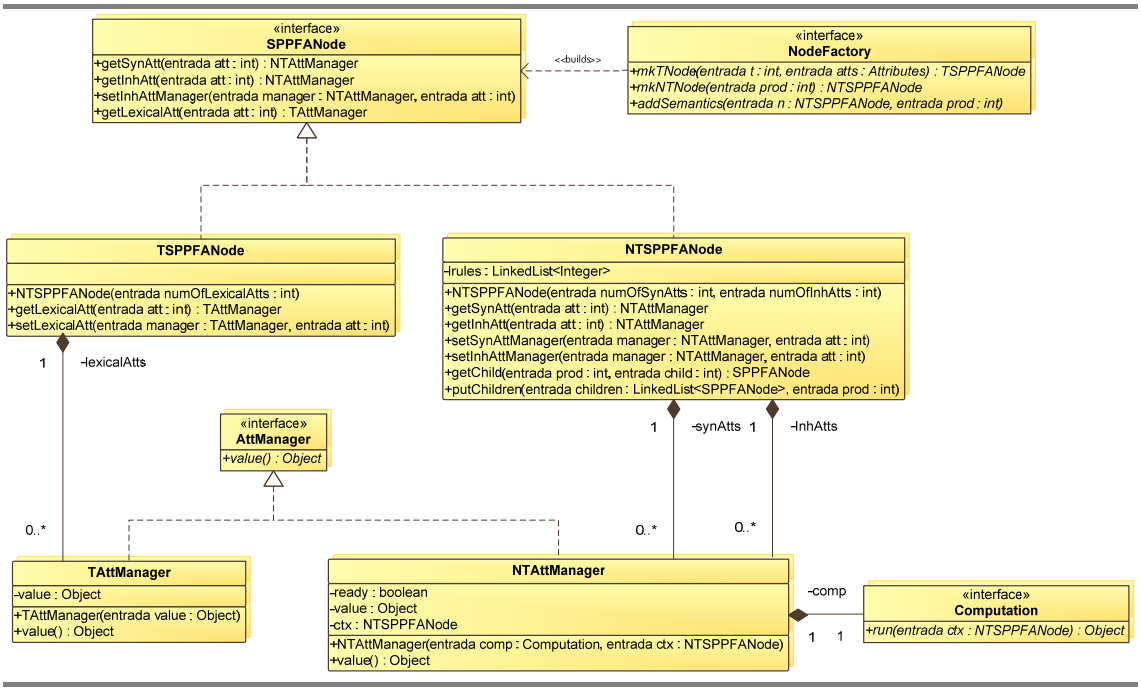


Figura 4.3.2. Diagrama de clases del modelo SPPFA.

La Figura 4.3.2 muestra el diagrama de clases del modelo SPPFA. La interface SPPFANode introduce el tipo base para los nodos del SPPFA. De esta forma:

- La clase *TSPPFANode* representa los nodos SPPFA terminales. El método *setLexicalAtt()* permite fijar el valor de un atributo léxico, mientras que el método *getLexicalAtt()* permite consultar el gestor de dicho valor.

- La clase *NTSPPFANode* implementa los nodos SPPFA no terminales. El método *putChildren()* permite añadir una nueva lista de hijos como resultado de reducir una producción. El método *getChild()* permite consultar un hijo concreto. Los métodos *setSynAttManager()* y *setInhAttManager()* permiten añadir, respectivamente, los gestores de valores para atributos sintetizados y atributos heredados, mientras que los métodos *getSynAtt()* y *getInhAtt()* permiten recuperar dichos gestores.

Nótese que los métodos *getSynAtt()*, *setInhAttManager()*, *getInhAtt()* de la clase *NTSPPFANode*, y el método *getLexicalAtt()* de la clase *TSPPFANode*, se han declarado como métodos públicos en la interfaz *SPPFANode*. El motivo es permitir la generación de un código más limpio, sin necesidad de realizar *casting* en los cómputos. Por su parte, la interface *AttManager* establece la interface para los gestores de valores. Existen dos implementaciones de dicha interface:

- La clase *TAttManager* implementa los gestores para los atributos de los nodos terminales. Se instancia internamente, al construir la clase *TSPPFANode*, con el valor del atributo léxico.

- La clase *NTAttManager* implementa los gestores para los atributos de los nodos no terminales. Estos gestores se construyen con una *computación*, objeto de tipo *Computation*, que encapsula el procedimiento de cómputo, y con el nodo de contexto sobre el que aplicar dicho procedimiento. El procedimiento de cómputo en sí debe implementarse como el método *run()* asociado a *Computation*.

Por último, la interface *NodeFactory* caracteriza la fábrica de nodos. El generador deberá, por tanto, generar una implementación adecuada de dicha interface para cada gramática. El método *mkTNode()* es el método constructor de los nodos terminales. Dicho método acepta como argumentos el código de un terminal, y una tabla con los atributos léxicos de dicho no terminal (instancia de clase *org.xml.sax.Attributes*), y devuelve el nodo construido. El método *mkNTNode()* es el análogo para nodos no terminales. En este caso, únicamente es necesario proporcionarle el código del no terminal. Por último, el método *addSemantics()* crea las computaciones necesarias para los atributos sintetizados de un nodo, y los heredados de sus hijos en una de sus listas de hijos. Para ello, recibe como parámetro el nodo y un identificador de producción.

4.3.3 Detalles de Implementación

Como en la versión XLOP 1.0, la extensión realizada procesa la especificación XLOP para obtener el modelo XLOP y una tabla de símbolos con información adicional recabada durante el proceso de análisis de dicha especificación. Mediante la construcción del autómata LALR(1) a partir del modelo XLOP, se obtienen las tablas de análisis. Por último, a través de la implementación del parser SAX se obtienen los distintos elementos XML del documento XML que se analiza. Con todos estos elementos, el algoritmo GLR puede construir correctamente el SPPF. Como hemos visto, es necesario construir nodos SPPFA y no nodos SPPF. Para ello, en los puntos del algoritmo GLR donde se invoca la construcción de un nodo SPPF se construye un nodo SPPFA a cambio, respetando el mismo tipo de nodo construido. El algoritmo resultante es una adaptación directa del presentado en la sección 2.3.7 (Figura 2.3.8 y Figura 2.3.9) del Capítulo 2. El cambio es directo, construyendo un nodo SPPFA terminal en la línea 69 y un nodo SPPFA no terminal en la línea 53 y 60 de las citadas figuras, invocando, para ello, los servicios apropiados de la factoría de nodos. El relleno de estas listas con gestores de valor para cada atributo se realiza invocando el método *addSemantics()* de dicha factoría. Más concretamente, en la factoría generada por el generador:

- Para un nodo SPPFA terminal, se llena cada posición de la lista de atributos léxicos con un gestor de valor de tipo *TAttManager*. El valor con el que se construye dicho gestor se obtiene del componente de información XML que le corresponde.

El orden de llenado de la lista respeta el orden de aparición de los atributos en la tabla de símbolos.

- Para un nodo SPPFA no terminal, cada posición se llena con un gestor de valor de tipo *NTAttManager*. La computación para dicho gestor se obtiene a partir de la ecuación del atributo. El nodo de contexto se fija siempre al nodo SPPFA correspondiente (para los atributos heredados, representará el padre adecuado a partir del cual evaluar la ecuación).

```
public interface AttManager {

   public abstract Object value();

}
```

```
public class NTAttManager implements AttManager {

   private Computation comp;
   private Object value;
   private boolean ready;
   private NTSPPFANode ctx;

   public NTAttManager(Computation comp, NTSPPFANode ctx) {
      this.comp = comp;
      this.ctx = ctx;
      this.ready = false;
   }

   @Override
   public Object value() {
      if (!ready) {
         value = comp.run(ctx);
         ready = true;
      }
      return value;
   }
}
```

```
public class TAttManager implements AttManager {

   private Object value;

   public TAttManager(Object value) {
      this.value = value;
   }

   @Override
   public Object value() {
      return value;
   }
}
```

Figura 4.3.3. Implementación de los gestores de valores.

La Figura 4.3.3 detalla la implementación de las clases relativas a los gestores de valores:

- Los gestores de tipo *TAttManager* se limitan a almacenar y devolver el valor del atributo.

- Los gestores de tipo *NTAttManager* mantienen, como ya se ha visto, información sobre si el valor se ha calculado ya o no se ha calculado. Cuando se solicita el valor, en caso de que no se haya calculado, se delega primeramente el cálculo en la computación que implementa el procedimiento de cómputo, comunicando a ésta el nodo de contexto.

```
public class <Nombre>NodeFactory implements NodeFactory {
  private <ClaseSemántica> semObj;
  private Computation[] comp;

  public <Nombre>NodeFactory() {
    comp = new Computation[<NúmeroExpresionesSemánticas>];
    semObj = new <ClaseSemántica>();

    comp[<NúmeroExpresiónSemántica>] = new Computation() {
      public Object run(NTSPPFANode ctx) {
        return <ExpresiónSemántica>;
      }
    };
    ...
    comp[<NúmeroExpresionesSemánticas - 1>] = ...
  }

<ExpresiónSemántica>  → XLOPHelper.invoke(semObj,"<NombreFunción>",new Object[]{ ( (<Arg> (,<Arg>)*)? })

<Arg> → (<AtributoSintetizado>|<AtributoHeredado>|<AtributoLéxico>|<ExpresiónSemántica>)

<AtributoSintetizado> → ctx.getChild(<Producción>,<Hijo>).getSynAtt(<Atributo>).value()

<AtributoHeredado> → ctx.getInhAtt(<Atributo>).value()

<AtributoLéxico> → ctx.getChild(<Producción>,<Hijo>).getLexicalAtt(<Atributo>).value()

  public NTSPPFANode mkNTNode(int nt){
    switch(nt) {
      case <NoTerminal>: return new NTSPPFANode(<NúmeroAtributosSintetizados>,<NúmeroAtributosHeredados>);
      ...
    }
    return null;
  }

  public TSPPFANode mkTNode(int t, Attributes atts) {
    TSPPFANode n = null;
    switch(t) {
      case <Terminal>: n = new TSPPFANode(<NúmeroAtributosLéxicos>); <MutadorLéxico>+ ; break;
      ...
    }
    return n;
  }

<MutadorLéxico> → n.setLexicalAtt(new TAttManager(atts.getValue("<NombreAtributoLéxico>")),<AtributoLéxico>);

  public void addSemantics(NTSPPFANode n, int prod) {
    switch(prod) {
      case <Producción>: (<MutadorAtributoSintetizado>|<MutadorAtributoHeredado>)+; break;
      ...
    }
  }
}

<MutadorAtributoSintetizado> → n.setSynAttManager(new NTAttManager(comp[<NúmeroExpresiónSemántica>],n),
                                <AtributoSintetizado>);

<MutadorAtributoHeredado> → n.getChild(prod,<Hijo>).
                    setInhAttManager(new NTAttManager(comp[<NúmeroExpresiónSemántica>],n),<AtributoHeredado>);
```

Figura 4.3.4. Plantilla de generación de código de la factoría de nodos.

Por último, es interesante examinar la plantilla que siguen las factorías de computaciones generadas por el generador. La Figura 4.3.4 muestra dicha plantilla. De esta forma:

- Para cada ecuación se crea una computación, que se asigna, en el constructor, a un elemento adecuado de un array de computaciones, con tantos elementos como ecuaciones semánticas. Así mismo, se crea una instancia de la clase semántica, que se asigna también a un campo privado de dicha factoría.

- El método *mkTNode()* se estructura como un *switch* sobre los códigos de terminales. En cada caso, se construye el nodo, fijando el tamaño de su lista de atributos léxicos en el constructor, y, seguidamente, se fijan los valores de los atributos léxicos.

- El método *mkNTNode()* se estructura también como un *switch*, esta vez sobre los no terminales. Para cada no terminal se construye un nodo, fijando de manera apropiada el tamaño de sus listas de atributos sintetizados y heredados.

- Finalmente, el método *addSemantics()* se estructura como un tercer *switch*, esta vez sobre los códigos de las producciones. Para cada producción y para cada ecuación en dicha producción, se crean gestores de valores adecuados, seleccionando, así mismo, las computaciones apropiadas (recuérdese que éstas están asignadas a campos de la clase), y estos gestores se instalan en los atributos que corresponden, en los nodos que corresponden.

```
   public NTSPPFANode mkNTNode(int nt){
      switch(nt) {
         case 0: return new NTSPPFANode(1,0); //A
         case 1: return new NTSPPFANode(2,1); //B
         case 2: return new NTSPPFANode(1,0); //C
      }
      return null;
   }

   public TSPPFANode mkTNode(int t, Attributes atts) {
      TSPPFANode n = null;
      switch(t) {
         case 0: n = new TSPPFANode(1); //#pcdata
                 n.setLexicalAtt(new TAttManager(atts.getValue("text")),0); break;
         case 1: n = new TSPPFANode(1); //<b>
                 n.setLexicalAtt(new TAttManager(atts.getValue("id")),0); break;
      }
      return n;
   }

   public void addSemantics(NTSPPFANode n, int prod) {
      switch(prod) {
         case 0: n.setSynAttManager(new NTAttManager(comp[0],n),0); //A ::= <a> B </a>
                 n.getChild(prod,1).setInhAttManager(new NTAttManager(comp[1],n),0); break;
         case 1: n.setSynAttManager(new NTAttManager(comp[2],n),0); break; //B ::= <b> #pcdata </b>
         case 2: n.setSynAttManager(new NTAttManager(comp[3],n),1); break; //B ::= C
         case 3: n.setSynAttManager(new NTAttManager(comp[4],n),0); break; //C ::= <b> #pcdata </b>
      }
   }
}
```

Figura 4.3.5. Ejemplo de factoría de nodos. Métodos mkNTNode(), mkTNode() y addSemantics().

La Figura 4.3.5 y la Figura 4.3.6 muestran tal implementación para la gramática de la Figura 4.2.3d. Obsérvese que, en los métodos *run()* de las computaciones, dado que la versión actual del modelo de evaluación no considera tipos, las funciones semánticas se invocan utilizando reflexión con la ayuda del método *invoke()* de la clase auxiliar *XLOPHelper*.

```
import org.xml.sax.Attributes;

public class ExampleNodeFactory implements NodeFactory {

   private ExampleSemanticClass semObj;
   private Computation[] comp;

   public ExampleNodeFactory() {
      comp = new Computation[5];
      semObj = new ExampleSemanticClass();
      comp[0] = new Computation() {
         public Object run(NTSPPFANode ctx) {
            return XLOPHelper.invoke(semObj,"f",
               new Object[]{ctx.getChild(0, 1).getSynAtt(1).value(),
                            ctx.getChild(0, 1).getSynAtt(0).value()});
         }
      };
      comp[1] = new Computation() {
         public Object run(NTSPPFANode ctx) {
            return ctx.getChild(0,1).getSynAtt(0).value();
         }
      };
      comp[2] = new Computation() {
         public Object run(NTSPPFANode ctx) {
            return ctx.getChild(1,1).getLexicalAtt(0).value();
         }
      };
      comp[3] = new Computation() {
         public Object run(NTSPPFANode ctx) {
            return XLOPHelper.invoke(semObj,"g",
               new Object[]{ctx.getChild(2,0).getSynAtt(0).value(),
                            ctx.getInhAtt(0).value()});
         }
      };
      comp[4] = new Computation() {
         public Object run(NTSPPFANode ctx) {
            return ctx.getChild(3,0).getLexicalAtt(0).value();
         }
      };
   }
```

Figura 4.3.6. Ejemplo de factoría de nodos. Atributos de la clase y constructora.

4.4 A modo de conclusión

Este capítulo ha presentado el modelo de ejecución para gramáticas de atributos multivista y la extensión de XLOP para soportar dicho modelo. La versión inicial de XLOP posee un generador de código que traduce las especificaciones XLOP en especificaciones CUP, en las que se aplica un mecanismo de evaluación retardado. En este proyecto de máster, el generador original se ha conservado, y se ha incluido un nuevo generador que produce una implementación del algoritmo GLR y del nuevo modelo de evaluación. Este modelo se basa en un mecanismo de evaluación dirigido por demanda, cuyo desarrollo se ha divido en:

- La determinación de un modelo SPPFA construido a partir del modelo SPPF generado por el algoritmo GLR. Este modelo es un enriquecimiento del modelo SPPF adaptado para su integración con el mecanismo de evaluación dirigido por demanda.

- La formulación de un mecanismo de instanciación del modelo SPPFA mediante ligeras modificaciones en el algoritmo GLR.

- El desarrollo del mecanismo de evaluación dirigido por demanda y su integración en el modelo SPPFA.

- La implementación concreta del modelo de evaluación y los cambios en el generador de XLOP que son necesarios para el correcto funcionamiento.

Este capítulo pues, ha mostrado cómo se puede incluir el soporte a las gramáticas de atributos multivista en el entorno XLOP de una manera práctica, sencilla y eficiente; y cómo dicho entorno es capaz de evolucionar para satisfacer nuevos requisitos de la misma manera. Así mismo, ha demostrado la factibilidad práctica de soportar dicho modelo en XLOP como componente básico para la especificación modular de tareas de procesamiento de documentos XML.

Capítulo 5
Conclusiones y trabajo futuro

5.1 Conclusiones

Este proyecto ha propuesto el uso de técnicas dirigidas por lenguajes para el desarrollo de aplicaciones de procesamiento XML. Más concretamente, la propuesta propugna utilizar gramáticas de atributos para especificar las tareas de procesamiento, de tal forma que los programas que implementan dichas tareas puedan generarse automáticamente a partir de estas especificaciones. Así mismo, en este proyecto se ha investigado en mecanismos que faciliten la modularización de dichas especificaciones, así como en los requisitos en tiempo de ejecución que precisan dichos mecanismos. A continuación se discuten con más detalle los principales resultados obtenidos, comparándolos con otros enfoques alternativos a los que se ha hecho alusión también en este proyecto.

5.1.1 XLOP frente a marcos genéricos de procesamiento

Tal y como se ha comentado en el Capítulo 2, uno de los enfoques más comúnmente extendido para el procesamiento de documentos XML es utilizar un marco genérico de procesamiento (p.ej., DOM, SAX o STaX). Como ya se comentó en dicho capítulo, la ventaja principal de dicho enfoque es contar con toda la flexibilidad de un lenguaje de programación de propósito general. No obstante, su principal desventaja es consecuencia directa de su generalidad: son marcos orientados al metalenguaje XML, en lugar de a sus aplicaciones específicas (p.ej., SVG, XHTML, etc.). De esta forma, los conceptos que manejan son conceptos relativos a XML y no a cada dominio de aplicación concreto. Existe, por tanto, una distancia considerable entre dichos conceptos y los conceptos del dominio, que se ve reflejada en los programas que procesan los documentos. Como consecuencia, dichos programas son difíciles de construir, y también difíciles de mantener.

Por su parte, XLOP propone un enfoque orientado al dominio, ya que las gramáticas XLOP caracterizan aplicaciones XML concretas (es decir, vocabularios XML concretos). De esta forma, XLOP evita la distancia conceptual entre los conceptos del marco y los conceptos del dominio intrínseco a los marcos genéricos. Por otra parte, dado que XLOP (y, en general, el enfoque basado en gramáticas de atributos) no norma la naturaleza de las funciones semánticas, sino que éstas pueden desarrollarse externamente (en el caso de XLOP, en Java), XLOP no sacrifica la generalidad propia de los marcos genéricos.

Así mismo, otra diferencia fundamental entre XLOP y los marcos genéricos de procesamiento es que los programas generados a partir de especificaciones XLOP son

consistentes con la gramática de la aplicación XML concreta (de hecho, están dirigidos por una sintaxis equivalente a la caracterizada por dicha gramática). Los marcos de procesamiento genérico, sin embargo, no implican en ningún momento dicha gramática, sino que es tarea del desarrollador asegurar que sus programas procesan los documentos que deben procesar. Esto es incluso cierto para aquellos documentos que hayan sido validados previamente por un parser, ya que, en estos marcos, la lógica específica está desacoplada del proceso de análisis (p.ej., en el modelo DOM, el parser construye el árbol DOM, y se lo *pasa* al componente que lo procesa; dicho componente puede, sin embargo, solicitar el tipo de elemento *Dri**ección***, aun cuando en la gramática el identificador de elemento sea *Dirección*).

5.1.2 XLOP frente a las propuestas de vinculación de datos

Tal y como se ha comentado en el Capítulo 2, las propuestas de vinculación de datos transforman la gramática documental en un conjunto de clases para representar los documentos que se ajustan a dicha gramática. De esta forma, dichas propuestas evitan la distancia conceptual intrínseca a los marcos genéricos de procesamiento. Así mismo, dichas propuestas son *conscientes* de la gramática documental. No obstante, el proceso de vinculación de datos, al operar directamente sobre dicha gramática, no es capaz de adaptarse a los requerimientos estructurales particulares de cada tarea de procesamiento concreta. De hecho, aunque este tipo de sistemas ofrezcan capacidades para configurar el proceso de vinculación (la denominadas *especificaciones de vinculación*; véase [Birbeck et al. 2001] para más detalles), dichas capacidades son limitadas, restringiéndose a aspectos tales como los nombres elegidos por las clases, cuándo representar un sub-elemento como una clase o un atributo, etc. Como consecuencia, este tipo de enfoques son apropiados para tipos de documentos simples, pero no para documentos con modelos de contenidos complejos o que intercalen contenidos textuales con contenidos estructurados.

XLOP comparte con las propuestas de vinculación de datos su naturaleza generativa. Sin embargo, en lugar de depender de un mecanismo de interpretación estructural pre-establecido, como ocurre con las propuestas de vinculación de datos, XLOP permite adaptar la estructura de los documentos a los requisitos de procesamiento de forma completamente flexible, y dependiente de cada lenguaje específico, de cada tarea de procesamiento, e, incluso, y en base a las extensiones propuestas en este proyecto, de cada aspecto concreto de cada tarea de procesamiento. El precio a pagar es un mayor esfuerzo de diseño, al tener que idear gramáticas apropiadas para cada lenguaje, tarea y aspecto. No obstante, debe tenerse en cuenta que dicho esfuerzo se realizará siempre de una u otra forma (p.ej., cuando se recorre un documento de facturación para recolectar información de los clientes, en dicho recorrido se está teniendo en cuenta de manera implícita la estructura sintáctica de dicho documento).

Así mismo, aquí también son aplicables las mismas consideraciones respecto al procesamiento dirigido por lenguajes que se han realizado en relación con los marcos genéricos de procesamiento: al final, el procesamiento residirá en una lógica específica desacoplada de la gramática del lenguaje. Aunque ahora se evitan errores del tipo de los

aludidos anteriormente, tampoco será posible asegurar que el procesamiento es consistente con la estructura gramatical del lenguaje, sobre todo cuando se ven implicados modelos de contenidos complejos o mixtos.

5.1.3 XLOP frente a los enfoques específicos

Tal y como se ha indicado en el Capítulo 2, el uso de enfoques específicos al procesamiento de documentos XML, como XSLT para especificar transformaciones, es interesante cuando la tarea de procesamiento se encuadra en el tipo de tarea abordado por el enfoque. Este hecho es consecuencia directa de que dichos enfoques vienen normalmente acompañados de lenguajes específicos de dominio, lo que facilita enormemente la realización de la tarea [Deursen et al. 2000; Mernik et al. 2005]. No obstante, la principal limitación de este tipo de enfoques es, paradójicamente, consecuencia de su principal ventaja: su especificidad. Cuando la tarea a realizar no se ajusta exactamente al tipo de tarea abordada por el enfoque específico, la solución es difícil de formular, anti-natural, o, simplemente, no puede encontrarse.

XLOP comparte con los enfoques específicos el brindar también un lenguaje específico de dominio, aunque esta vez dirigido a especificar la capa lingüística de las aplicaciones de procesamiento. De esta manera, XLOP comparte con dichos enfoques la ventaja de utilizar lenguajes específicos orientados a resolver un tipo de tarea concreta. En el caso de XLOP, el lenguaje se basa en el formalismo de las gramáticas de atributos, y la tarea es la de procesamiento de lenguajes de cadenas. Debido a la genericidad de la tarea abordada por XLOP, para tipos de tareas más específicas para los que existan enfoques especialmente diseñados para los mismos, el coste de aplicar XLOP será mayor. No obstante, XLOP proporciona también la flexibilidad asociada con un enfoque genérico, lo que, junto con el carácter específico de su lenguaje de especificación, permite conseguir un equilibrio razonable sobre un amplio espacio de tareas.

5.1.4 XLOP frente a enfoques basados en esquemas de traducción

En el Capítulo 2 se han referido algunas propuestas al procesamiento de documentos XML basadas en esquemas de traducción (p.ej., ANTXR y RelaxNGCC). Tal y como se comentó en dicho capítulo, un esquema de traducción consiste en una gramática incontextual con acciones semánticas asociadas. Dichas acciones se ejecutan durante el proceso de análisis sintáctico, por lo que están acopladas con dicho proceso. De esta forma, el desarrollador que escribe un esquema de traducción debe ser consciente del algoritmo de análisis que se va a aplicar, con el fin de asegurar el correcto orden de ejecución de las acciones semánticas. Por su parte, una gramática de atributos es un formalismo de más alto nivel, ya que la *ejecución* de las ecuaciones semánticas no depende del algoritmo de análisis, sino de las dependencias entre los atributos. De esta forma, el desarrollador no tiene porque ser consciente del algoritmo de análisis, lo que convierte a las gramáticas de atributos en formalismos más declarativos y de

más alto nivel que los esquemas de traducción, y, por tanto, a XLOP en una herramienta de más alto nivel que ANTXR o RelaxNGCC.

Por su parte, el enfoque promovido por RelaxNGCC difiere también de XLOP en otro aspecto. RelaxNGCC adopta la propia gramática documental como la estructura sintáctica que dirige la especificación (de hecho, es una extensión del lenguaje de esquema RelaxNG). Por su parte, un aspecto fundamental de XLOP es propugnar el ajuste de la estructura sintáctica al procesamiento a llevar a cabo. De esta forma, las gramáticas incontextuales subyacentes a una especificación XLOP han de ser equivalentes a la gramática documental, pero no tienen por qué *ser* dicha gramática documental. La gramática documental y las gramáticas incontextuales de una especificación XLOP tienen propósitos diferentes: la primera define el lenguaje de cara al autor de los documentos (lo mismo que una descripción EBNF o basada en grafos sintácticos define un lenguaje de programación de cara al programador); la segunda define el lenguaje de cara al desarrollador de la aplicación de procesamiento (lo mismo que una gramática LALR(1) de un lenguaje de programación es una gramática apropiada para el desarrollador del compilador del mismo). En este sentido, XLOP se aproxima más al enfoque sostenido por ANTXR (que no es más que un preprocesador de la herramienta de generación de *parsers* ANTLR).

5.1.5 XLOP frente a otros enfoques basados en gramáticas de atributos

El Capítulo 2 también ha presentado algunos enfoques al procesamiento de documentos XML basados en gramáticas de atributos. La mayor parte de dichos enfoques (p.ej., los descritos en [Feng & Wakayama 1993], [Neven 2005], [Gançarski et al. 2006]) utilizan las gramáticas de atributos como mecanismos auxiliares para expresar la semántica de otro tipo de tareas (normalmente, tareas de consultas a documentos). La idea básica consiste en describir cómo formalizar dichas tareas en términos de gramáticas de atributos. De esta forma, en dichas propuestas las gramáticas de atributos juegan un papel completamente auxiliar. De hecho, buena parte de los esfuerzos de dichas propuestas están orientados a describir cómo fusionar el formalismo EBNF intrínseco a las gramáticas documentales con el formalismo de las gramáticas de atributos (que tradicionalmente requiere gramáticas en formato BNF).

En contraposición a dichas propuestas, XLOP defiende el rol de las gramáticas de atributos como mecanismos centrales en el desarrollo de programas que procesan documentos XML. Así mismo, XLOP propugna la formulación de gramáticas BNF equivalentes a las gramáticas documentales como un aspecto central de dicho desarrollo, en lugar de tratar de amoldar el formalismo EBNF a las peculiaridades de las gramáticas de atributos.

Una propuesta similar a XLOP es la realizada por [Psaila & Crespi-Reghizzi 1999]. Sin embargo, en dicha propuesta, en lugar de fomentar la formulación de gramáticas específicas para cada procesamiento, se decidió desacoplar la especificación de la gramática, asociando las ecuaciones semánticas en función de pares *padre – hijo* en los árboles documentales como

una forma de puentear el problema de reconciliar la notación EBNF con el formalismo de las gramáticas de atributos. Por su parte, las propuestas de [Nishimura & Nakano 2005] y [Nakano 2004], siendo similares en espíritu a XLOP, se diferencian de nuestro enfoque en que dependen de transformaciones de flujos XML a árboles, especificadas a su vez mediante gramáticas de atributos, en lugar de realizadas mediante algoritmos específicos de análisis sintáctico.

5.1.6 Mecanismos de modularidad en XLOP frente a otros enfoques

Tal y como ya se ha indicado, el mecanismo de modularidad básico de XLOP, basado en la descomposición de aspectos semánticos, es similar a la propuesta clásica de [Kastens & Waite 1994]. No obstante, la característica distintiva de XLOP es permitir variar también la sintaxis de los distintos aspectos asociados a una misma tarea. El resultado es, como ya hemos indicado en este proyecto, las gramáticas de atributos multivista [Temprado 2010b]. Hasta donde alcanza nuestro conocimiento, dicha alternativa no ha sido explorada en otros entornos basados en gramáticas de atributos.

Por su parte, el mecanismo de espacios de nombres en XLOP no debe confundirse con el mecanismo de espacio de nombres en XML [Namespaces 2009]. Efectivamente, en XLOP dicho mecanismo permite que los no terminales de cada sublenguaje pueden residir en su propio espacio de nombres, lo que facilita la gestión de nombres de no terminales, y, por tanto, evita posibles conflictos entre las distintas reglas asociadas con cada sublenguaje. Por su parte, en XML los espacios de nombres se utilizan para evitar conflictos entre tipos de elementos y de atributos XML, mientras que en XLOP se utilizan para evitar conflictos entre reglas gramaticales. De hecho, XLOP *no* introduce mecanismos específicos para tratar con los espacios de nombres de los elementos, ya que el propósito de cada elemento se podrá deducir de las reglas gramaticales en las que éste aparece. Así, por ejemplo, una etiqueta *Dirección* que aparezca en una regla para un no terminal *Persona* se referirá a la dirección de una persona, mientras que si dicha etiqueta aparece en una regla para el no terminal *Maquina*, se referirá probablemente a la dirección de un ordenador en una red de ordenadores.

Así mismo, la introducción de múltiples vistas semánticas en XLOP permite asociar múltiples estructuras con un mismo documento. Este hecho recuerda al uso de *marcado concurrente* en SGML [Goldfarb 1991]. En SGML, dicho marcado permite marcar un mismo documento, que es susceptible de exhibir distintas estructuras, con respecto a diferentes DTDs. En cierta forma, el propósito de las vistas semánticas es similar. La diferencia es que, ahora, las estructuras se deducen automáticamente a partir del marcado (en lugar de ser anotadas sobre el documento por el autor), que dichas estructuras, los árboles de análisis sintáctico, tienen un propósito operacional (en lugar de un propósito descriptivo, como el añadido de marcas a los documentos en SGML o en XML), y que, además, existe una manera de integrarlas en una estructura común: el SPPF. Dicha estructura, además, puede instrumentarse a partir del componente semántico de la gramática multivista, para dar lugar a

un SPPFA, sobre el que puede realizarse la evaluación por demanda de los atributos semánticos.

5.1.7 Aplicabilidad práctica de XLOP

Aunque las experiencias realizadas con XLOP no permiten aún contrastar cuantitativamente la aplicabilidad práctica de XLOP, las experiencias realizadas en relación con el sistema experimental <e-Tutor> [Temprado 2010a], que han sido descritas en el Capítulo 3, permiten analizar cualitativamente las ventajas del entorno durante el desarrollo, y sobre todo, durante el mantenimiento de aplicaciones de procesamiento de documentos XML complejas. En particular, el entorno propugna la separación explícita de los aspectos relativos al procesamiento directo de los documentos y los relativos a la lógica específica de la aplicación, y facilita la co-evolución y la evolución separada de ambos aspectos. Además, al proporcionar un lenguaje especialmente diseñado para ello, facilita especialmente la producción y el mantenimiento de la capa lingüística.

No obstante, también durante el desarrollo de <e-Tutor> hemos podido comprobar que el enfoque dirigido por lenguajes adoptado en XLOP presupone conocimientos y habilidades que pueden no estar al alcance del desarrollador medio. Efectivamente, desarrollar aplicaciones con XLOP es similar a desarrollar procesadores / traductores para lenguajes informáticos, lo que implica conocimientos específicos que se adquieren en etapas avanzadas de la formación universitaria en Informática. Así mismo, el uso de gramáticas de atributos no tiene porque resultar sencillo a la mentalidad imperativa dominante entre la mayor parte de los desarrolladores. Por otra parte, incluso para desarrolladores expertos en el uso de gramáticas de atributos, la proliferación de ecuaciones superfluas (p.ej., ecuaciones de copia) puede resultar engorrosa. De hecho, durante nuestros experimentos con <e-Tutor> esto nos llevó a proponer un estilo de especificación basado en el mantenimiento explícito de estado en la clase semántica, y en el uso de atributos que representen eventos asociados con los cambios de dicho estado. Este aspecto, sin embargo, requiere un mayor análisis, a fin de encontrar formas de acercar el formalismo al desarrollador medio.

Por otra parte, las extensiones realizadas en este proyecto de máster se han visto motivadas también por las experiencias realizadas sobre <e-Tutor>. No obstante, la evaluación del grado de aplicabilidad de dichas propuestas está aún en una fase muy inicial, limitándose a ejemplos *de juguete*, como el de las expresiones aritméticas presentado en el Capítulo 3. Aunque los mecanismos propuestos parecen razonables, se estima necesario seguir trabajando en relación con su aplicabilidad práctica a fin de refinar los mismos y determinar en qué grado facilitan la aplicación práctica de la totalidad del entorno XLOP.

5.2 Trabajo Futuro

Como resultado de la realización de este proyecto es posible proponer distintas líneas de trabajo futuro, algunas de las cuáles se detallan a continuación.

5.2.1 Comprobación automática de las restricciones contextuales de las gramáticas multivista

Como se ha descrito en el Capítulo 3, las gramáticas multivista vienen acompañadas de un conjunto de restricciones contextuales cuyo cumplimiento asegura su correcto comportamiento en tiempo de ejecución. La más básica es que las gramáticas incontextuales subyacentes han de ser equivalentes entre sí. Así mismo, dichas gramáticas deben ser equivalentes a la gramática documental del lenguaje. En el caso general, no será posible comprobar automáticamente dichas restricciones, por lo que nuestra propuesta impone que las subgramáticas asociadas a los no terminales de núcleo sean no-autoembebibles. Esta restricción, que puede comprobarse automáticamente, asegura, además, que los lenguajes generados por estas subgramáticas sean regulares (considerados, en cada subgramática, los no terminales del núcleo como terminales). Es más, existen algoritmos para construir autómatas finitos reconocedores equivalentes, lo que, a su vez, permite comprobar las equivalencias de las subgramáticas en cada vista, y de éstas con los correspondientes modelos de contenidos, condición suficiente para que las vistas sean equivalentes entre sí, y equivalentes con la gramática documental. Por otra parte, es necesario tratar con el hecho de que una misma porción de documento debe analizarse simultáneamente a través de varias gramáticas. Desde un punto de vista formal, la reunión de dichas gramáticas dará lugar a una gramática equivalente, aunque ambigua. Por tanto, nuestra propuesta adopta un método de análisis sintáctico capaz de tratar eficientemente con gramáticas ambiguas: el método GLR que se deriva de los trabajos de Tomita en procesamiento de lenguaje natural. No obstante, nuestra propuesta utiliza dicho algoritmo, más que como mecanismo para tratar con la ambigüedad, como mecanismo para llevar a cabo eficientemente el análisis simultáneo respecto a las distintas vistas. De hecho, si se exige que la gramática asociada a cada vista sea LALR(1) (y, por tanto, LR(1)), los no terminales del núcleo se corresponderán con nodos de empaquetamiento en el bosque de análisis sintáctico resultante, y, por tanto, reflejarán fielmente el concepto de ser los *puntos de unión* entre las distintas vistas. Por último, el concepto de *gramática de atributos multivista* implica que, desde un punto de vista metalingüístico, sea necesario flexibilizar las restricciones habituales respecto a la definición de atributos. Por una parte, dado un símbolo del núcleo, cada uno de sus atributos sintetizados ha de definirse una única vez en cualquiera de las reglas para dicho símbolo en cualquiera de las vistas. Por otra parte, cada vista ha de centrarse en un conjunto de sus atributos heredados, debiéndose definir estos en todas aquellas producciones de la vista en las que el símbolo ocurre en las partes derechas. Este hecho nos ha llevado a introducir en XLOP mecanismos explícitos para declarar y relacionar vistas entre sí. Actualmente estamos trabajando en la implementación de algoritmos eficientes para comprobar automáticamente todas estas restricciones contextuales.

En particular, un aspecto aún no resuelto es comprobar de manera eficiente las restricciones relativas al carácter LALR(1) de cada vista. Pensamos, así mismo, que la resolución de este aspecto tendrá también un impacto importante de cara a la modularidad, al permitir la composición, no de especificaciones, sino de los procesadores generados a partir de las mismas. Para ello estamos considerando actualmente trabajos relativos a la composición de tablas de análisis ascendente, como los descritos en [Schwerdfeger & Van-Wyk 2009] y [Xiaoqing et al. 2010].

5.3 Aumento de la eficiencia del mecanismo de evaluación de las gramáticas multivista

Una de las principales características del método de evaluación de XLOP 1.0 es permitir intercalar los procesos de análisis y evaluación. De esta forma, no es necesario esperar a finalizar el análisis para comenzar el proceso de evaluación, lo que tiene importantes implicaciones de cara al procesamiento asíncrono de documentos XML (p.ej., documentos que llegan a través de un *stream* en un escenario de procesamiento distribuido). Así mismo, dado que el espacio ocupado por aquellos atributos que ya no sean necesarios puede liberarse, bajo ciertas circunstancias y para cierto tipo de gramáticas, XLOP 1.0 garantiza una ejecución que requiere una cantidad constante de memoria auxiliar, independientemente de la anchura de los documentos procesados. Para documentos muy grandes (p.ej., los que surgen en dominios de aplicación como los de el procesamiento de datos astronómicos, o en bioinformática), esto puede suponer una clara ventaja frente a métodos basados en la construcción explícita del árbol de análisis sintáctico. No obstante, el modelo de evaluación propuesto en este proyecto no comparte dicha característica, al obligar a construir explícitamente el SPPFA antes de comenzar la evaluación de sus atributos. De esta forma, como línea de trabajo futuro surge el formular un modelo de evaluación *on-line* que, como ocurre con XLOP 1.0, permita intercalar los procesos de análisis y evaluación, y permita mejorar la eficiencia en memoria. En particular, la optimización de la GSS ya adelantada en el Capítulo 2 es uno de los aspectos que deberán ser estudiados como consecuencia de esta línea de investigación.

5.3.1 Tipado de las especificaciones XLOP

Actualmente, el lenguaje de especificación no está tipado. Esto trae como consecuencia un decremento en la usabilidad del mismo, ya que los errores de tipos (p.ej., invocar una función semántica con argumentos con tipos erróneos) se detectan en tiempo de ejecución. De esta forma, como trabajo futuro planteamos también la inclusión de un mecanismo de tipado en XLOP. Dicho mecanismo puede basarse inicialmente en el tipado explícito de los atributos, y en la comprobación de tipos en las expresiones semánticas utilizando el mecanismo de reflexión de Java. No obstante, también se prevé incluir un mecanismo de inferencia de tipos, con el fin de flexibilizar dicho aspecto.

5.3.2 Depuración de XLOP

Durante nuestras experiencias con XLOP hemos comprobado que uno de los aspectos que pueden disminuir su usabilidad es la carencia de mecanismos apropiados para depurar las especificaciones. Este aspecto es particularmente delicado, debido a que el flujo asociado al proceso de evaluación de atributos no está explícitamente determinado, sino que es consecuencia de las dependencias entre atributos. Esta característica, que supone una ventaja clara desde el punto de vista de facilitar el proceso de especificación, también dificulta la depuración. De esta forma, actualmente estamos trabajando en el desarrollo de un entorno visual de depuración para XLOP 1.0, que permite visualizar la manera en la que se *construye* el árbol de análisis sintáctico, así como la forma y el orden en la que se realiza el cómputo de los atributos. En el futuro planteamos también aplicar un enfoque similar a las gramáticas multivista propuestas en este proyecto.

5.3.3 Inclusión de patrones de atribución en XLOP

Como se ha comentado en la sección anterior, uno de los principales defectos de la especificación basada en gramáticas de atributos es la necesidad de expresar explícitamente todas las propagaciones de atributos, aún cuando los valores de los atributos propagados no cambien. Una manera de evitar este problema es utilizar los denominados *patrones de atribución* [Kastens & Waite 1994]. Estos patrones permiten definir reglas de propagación por defecto para ciertos atributos. Un ejemplo típico es el de *atributo remoto*: un atributo heredado que se propaga a cada uno de los hijos, sin necesidad de expresar las correspondientes ecuaciones de copia. No obstante, algunos experimentos realizados en esta línea nos han convencido de que, si bien las especificaciones que resultan son más compactas, también son más difíciles de entender. Proponemos, por tanto, buscar soluciones a este problema, que pueden basarse en introducir otro tipo de metáfora diferente a la clásica de *las ecuaciones semánticas que computan valores para atributos semánticos*, aunque formalmente equivalente a ésta.

5.3.4 Entorno de desarrollo integrado para XLOP

Actualmente, y también en relación con XLOP 1.0, estamos construyendo un entorno integrado de desarrollo como un *plug-in* de Eclipse. Dicho entorno ofrece un editor sensible a la sintaxis, así como los mecanismos habituales de gestión de proyectos, e integra también la interfaz de generación y ejecución de aplicaciones descrita en [Martínez & Temprado 2009]. De esta forma, como siguiente paso en esta línea proponemos extender dicho entorno para que soporte las extensiones descritas en este proyecto de investigación.

5.3.5 Evaluación

Como última línea de trabajo, proponemos una evaluación cuantitativa detallada de los distintos aspectos del desarrollo de aplicaciones con XLOP. Dicha evaluación estará centrada en los siguientes ejes:

- Evaluación de la economía y flexibilidad expresiva. Esta evaluación se llevará a cabo mediante el desarrollo de aplicaciones de prueba adicionales, comparando las soluciones dadas por XLOP con soluciones desarrolladas utilizando otras alternativas.

- Evaluación de la eficiencia. La evaluación se llevará a cabo comparando la eficiencia de soluciones XLOP con soluciones dadas en función de otras alternativas. En este eje se compararán también modelos de evaluación de atributos alternativos.

- Evaluación de la usabilidad. Para ello se llevarán a cabo experimentos controlados de evaluación con usuarios reales. El experimento se dirigirá a estudiantes de últimos cursos de Ingenierías Informáticas y de los nuevos grados, así como a estudiantes del Máster en Investigación Informática.

Referencias

1. [Ablas 1991] Ablas, H. Attribute Evaluation Methods. En Ablas, H., Melinchar, B (eds.) Attribute Grammars, Application and Systems. Lecture Notes in Computer Science 545. Springer. 1991.

2. [Adobe 2007] Adobe Systems Inc. Postscript(R) Language Tutorial and Cookbook (APL). Addison-Wesley. 2007.

3. [Aho et al. 2007] Aho, A.V., Lam, M.S., Sethi, R., Ullman, J.D. Compilers: principles, techniques and tools (second edition). Addison-Wesley. 2007.

4. [Appel 1997] Appel, A.W. Modern Compiler Implementation in Java. Cambridge University Press. 1997.

5. [Birbeck et al. 2001] Birbeck, M et al. Professional XML 2nd Edition. WROX Press, Birminghan,UK. 2001.

6. [Bradley 2001] Bradley, N. The XML Companion (3rd Edition). Addison-Wesley. 2001.

7. [Bray et al. 2008] Bray, T., Paoli, J., Sperberg-McQueen, C.M., Maler, E., Yergeau, F. Extensible Markup Language (XML) 1.0 (Fifth Edition). W3C Recommendation. http://www.w3.org/TR/REC-xml. 2008.

8. [Bork 1985] Bork, A. Personal Computers for Education. Harper & Rows. 1985.

9. [Brownell 2002] Brownell, D. SAX2 – Procesing XML Efficiently with Java. O'Relley. 2002.

10. [Coombs et al. 1987] Coombs, J. H. Renear, A. H., DeRose, S. J. Markup Systems and the Future of Scholarly Text Processing. Communications of the ACM, 30 (11), 933-947. 1987.

11. [DeRemer 1969] DeRemer, F. Practical Translators for LR(k) Languages. Ph.D. Thesis. MIT, Cambridge, MA, 1969.

12. [Declarative Systems 1992] Declarative Systems. User Manual for the Linguist Translator-Writing System. Declarative Systems, Inc., Palo Alto, Calif. 1992.

13. [Deursen et al. 2000] Deursen A, Klint P, Visser J. Domain-Specific Languages: An Annotated Bibliography. ACM SIGPLAN Notices. June; 35(6): 26-36. 2000.

14. [DOM 2009] DOM. Document Object Model Technical Reports. W3C Recommendations. http://www.w3.org/DOM/DOMTR. 2009.

15. [Dueck & Cormack 1990] Dueck, G. D. P., Cormack, G. V. Modular attribute grammars. Comput. J. 33, 2, 164– 172. 1990.

16. [Earley 1968] Earley, J. An Efficient Context free Parsing Algorithm. PhD thesis, Computer Science Department, Carnegie-Mellon University, 1968.

17. [Emerson 1986] Emerson, S. Troff Typesetting for UNIX Systems. Prentice Hall. 1986.

18. [Feng & Wakayama 1993] Feng, A., Wakayama, T. A Grammar-based Transformation System for Structured Documents, Electronic Publishing, vol. 6, no. 4, pp. 361-372, 1993.

19. [Gançarski et al. 2006] Gançarski, A. L., Doucet, A., Henriques, P. R. Grammar-based Interactive System to Retrieve Information from XML Documents, IEE Proceedings-Software, vol. 153, no. 2, pp. 51-60, 2006.

20. [Ganzinger & Giegerich 1984] Ganzinger, H., Giegerich, R. Attribute Coupled Grammars, SIGPLAN Symposium on Compiler Construction, Montreal, Canada, pp. 157 - 170. 1984.

21. [Goldfarb 1981] Goldfarb, C. A generalized approach to document markup. ACM SIGPLAN Notices 16(6). 68-73. 1981.

22. [Goldfarb 1991] Goldfarb, C. The SGML Handbook. Oxford University Press. 1991.

23. [Goldberg & Robson 1983] Goldberg, A. Robson, D. Smalltalk-80. The Language and Its Implementation Addison-Wesley, Reading, Mass. 1983.

24. [Gosling et al. 2005] Gosling, J., Joy, B., Steele, G.L., Bracha, G. The Java Language Specification Third Edition. Addison Wesley. 2005.

25. [Gross et al. 1989] Gross, T., Zobel. A., Zoi.G, M. Parallel compilation for a parallel machine In Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation. ACM SIGPLAN Not. 24, 7, 91–100. 1989.

26. [Grossmann et al. 1984] Grossmann, R., Hutschenreiter, J., Lanwe, J., Lo Tzsch, J., Mager, K. DEPOT2a—Metasystem fur dle Analyse und Verarbeitung verbundener F'achsprachen. Anwenderhandbuch, Sektion Mathematlk, Technische Universität Dresden. 1984.

27. [Grune & Jacobs 2008] Grune, D., Jacobs, C.J.H. Parsing Tecniques – A Practical Guide 2nd Edition. Monographs in Computer Science. Springer. 2008.

28. [Hedin 1989] Hedin, G. An object-oriented notation for attribute grammars In Proceedings of the 3rd European Conference on Object-Oriented Programming (ECOOP '89), S. Cook, Ed. British Informatics Society Ltd., Nottingham, 329-345. 1989.

29. [Hopcroft 1979] Hopcroft, J.E., Ullman, J.D. Introduction to Automata Theory, Languages and Computation. Addison Wesley. 1979.

30. [Ibrahim 1989] Ibrahim, B. Software Engineering Techniques for CAL. Education & Computers 5, 215-222. 1989.

31. [Johnsson 1987] Johnsson, T. Attribute grammars as a functional programming paradigm In Proceedings of the Symposium on Functional Programming Languages and Computer Architecture. Lecture Notes in Computer Science, vol. 274, G Kahn, Ed. Springer-Verlag, New York, 154-173. 1987.

32. [Johnstone et al. 2004] Johnstone, A., Scott, E., Economopoulos, G. The grammar tool box: A case study comparing GLR parsing algorithms. In Proceedings of the 4th Workshop on Language Descriptions, Tools and Applications LDTA2004, G. Hedin and E. V. Wick, eds. Also in Electronic Notes in Theoretical Computer Science. Elsevier, New York. 2004.

33. [Jourdan et al. 1990] Jourdan, M., Parigot, D. Internals and externals of the FNC-2 attribute grammar system. In Attribute Grammars, Applications and Systems. Lecture Notes in Computer Science, vol 545. Springer-Veriag, New York, 485-506. 1991.

34. [Kastens & Waite 1994] Kastens, U., Waite, W. M. Modularity and Reusability in Attribute Grammars, Acta Informatica, 31(7), 601-627, 1994.

35. [Kawaguchi 2002] Kawaguchi, K. Flexible Data-Biding with RelaxNGCC. Extreme Markup Languages 2002, August 4-9, Montreal, Canada. 2002.

36. [Kay 2007] Kay, M (ed.). XSL Transformations (XSLT) Version 2.0. W3C Recommendation. http://www.w3.org/TR/xslt20. 2007.

37. [Koskimies et al. 1988] Koskimies, K., Nurmi, O., Paakki, J., Sippu, S. The design of a language processor generator. Softw. Pract. Exper. 18, 2, 107–135. 1988.

38. [Koskimies 1989] Koskimies, K. Software engineering aspects in language implementation. In Proceedings of the 2nd Workshop on Compder Compilers and High Speed Compilation. Lecture Notes in Computer Science, vol. 371, D. Hammer, Ed. Springer-Verlag, New York, 39-51. 1989.

39. [Koskimies & Paakki 1990] Koskimies, K., Paakki, J. Automating Language Implementation-A Pragmatic Approach, Ellis Horwood, Chichester, England. 1990.

40. [Knuth 1965] Donald E. Knuth: On the Translation of Languages from Left to Rigth Information and Control 8(6): 607-639. 1965.

41. [Knuth 1968] Knuth, D. E. Semantics of Context-free Languages. Mathematical System Theory 2(2), 127–145. Ver también Math. System Theory 5(1), 95–96. 1968.

42. [Lemke 1993] Lemke, I. Sander, G. Visualization of Compiler Graphs. Design report D 3.12.1-1, USAAR-1025-visual, ESPRIT Project #5399 Compare, Universität des Saarlandes, FB 14 Informatik, 1993.

43. [Lam et al. 2008] Lam, T.C., Ding, J.J., Liu, J.C. XML Document Parsing: Operational and Performance Characteristics. IEEE Computer 41(9), 30-37. 2008.

44. [Magnusson & Hedin] Magnusson E., Hedin G.: Circular reference attributed grammars - their evaluation and applications. Sci. Comput. Program. 68(1): 21-37. 2007.

45. [Maluszynski 1991] Maluszynski, J. Attribute grammars and logic programming: A comparison of concepts. In Attribute Grammars, Applications, and Systems. Lecture Notes in Computer Science, vol. 545. Springer-Verlag, New York, 330-357. 1991.

46. [Martínez & Temprado 2009] Martínez A., Temprado B. XLOP: XML Language-Oriented Processing. Proyecto fin de carrera. Universidad Complutense de Madrid. 2009.

47. [Maruyama et al. 2002] Maruyama, H. et al. XML and Java: Developing Web Applications. Addison-Wesley. 2002.

48. [Mernik et al. 2005] Mernik M, Heering J, Sloane AM. When and how to Develop Domain-Specific Languages. ACM Computing Surveys. December; 37(4): 316-344. 2005.

49. [Murata et al. 2005] Murata, M., Lee, D., Mani, M., Kawaguchi, K. Taxonomy of XML schema languages using formal language theory. ACM Transactions on Internet Technology 5(4), 660-704. 2005.

50. [Nakano 2004] Nakano, K. An Implementation Scheme for XML Transformation Languages Through Derivation of Stream Processors, in Programming Languages and Systems: Second Asian Symposium (APLAS'04), Taipei, Taiwan, pp. 74-90. 2004.

51. [Namespaces 2009] Namespaces. Namespaces in XML 1.0 (Third Edition). W3C recommendation, 2009.

52. [Nederhof & Sarbo 1996] Nederhof, M. J., Sarbo, J. J. Increasing the applicability of LR parsing. In Recent Advances in Parsing Technology, H.Bunt and M. Tomita, eds. Kluwer Academic, Amsterdam, the Netherlands, 35–57. 1996.

53. [Nederhof 2000] Nederhof, M. J. Regular Approximations of CFLs: A Gramatical View. In Bunt, H, M Nijholt, A (eds.): Advances in Probabilistic and other Parsing Technologies. Kluwer. 2000.

54. [Neven 2005] Neven, F. Attribute Grammars for Unranked Trees as a Query Language for Structured Documents, Journal of Computer and System Sciences, vol. 70, no. 2, pp. 221-257, 2005.

55. [Nishimura & Nakano 2005] Nishimura, S., Nakano, K. XML Stream Transformer Generation through Program Composition and Dependency Analysis, Science of Computer Programming, vol. 54, no. 2-3, pp. 257-290, 2005.

56. [Nozohoor-Farshi 1991] Nozohoor-Farshi, R. GLR parsing for e-grammars. In Generalized LR Parsing, M. Tomita, ed. Kluwer Academic, Amsterdam, the Netherlands, 60–75. 1991.

57. [Paakki 1995] Paakki, J. Attribute Grammar Paradigms – A High-Level Methodology in Language Implementation. ACM Computing Surveys, 27, 2, 196-255. 1995.

58. [Psaila & Crespi-Reghizzi 1999] Psaila, G., Crespi-Reghizzi, S. Adding Semantics to XML, in 2nd International Workshop on Attribute Grammars and their Applications (WAGA'99), Amsterdam, The Netherlands, pp. 113-132. 1999.

59. [Rebernak et al. 2006] Rebernak, D., Mernik, M., Henriques, P. R. et al. AspectLISA: an Aspect-oriented Compiler Construction Systems based on Attribute Grammars. 6th Workshop on Language Description Tools and Applications (LDTA'06), Vienna, Austria. 2006.

60. [Rekers 1992] Rekers, J. G. Parser generation for interactive environments. Ph.D. thesis, Universty of Amsterdam. 1992.

61. [Saraiva & Swierstra 1999] Saraiva, J., Swierstra, D. Generic Attribute Grammars. 2nd International Workshop on Attribute Grammars and their Applications (WAGA'99), Amsterdam, The Netherlands. 1999.

62. [Sarasa et al. 2008] Sarasa, A., Navarro, I., Sierra, J.L, Fernández-Valmayor, A. Building a Syntax Directed Processing Environment for XML Documents by Combining SAX and JavaCC. 3rd International Workshop on XML Data Management Tools & Techniques. DEXA'08. 2008.

63. [Sarasa et al. 2009a] Sarasa, A., Martínez-Avilés, A., Sierra, J.L., Fernández-Valmayor, A. A Generative Approach to the Construction of Application-Specific XML Processing Components. 35th Euromicro Software Engineering and Advanced Applications Conference. 2009a.

64. [Sarasa et al. 2009b] Sarasa, A., Sierra, J.L. Fernández-Valmayor, A. Procesamiento de Documentos XML Dirigido por Lenguajes en Entornos de E-Learning. IEEE RITA, en prensa. 2009b.

65. [Sarasa et al. 2009c] Sarasa, A., Sierra, J.L. Fernández-Valmayor, A.Processing Learning Objects with Attribute Grammars. 9th IEEE International Conference on Advanced Learning Technologies. 2009c.

66. [Sarasa et al. 2009d] Sarasa, A., Temprado, B., Sierra, J.L. Fernández-Valmayor, A. XML Language-Oriented Processing with XLOP. 5th International Symposium on Web and Mobile Information Services. 2009d.

67. [Sarasa et al. 2009e] Sarasa, A., Temprado-Battad, B., Martínez-Avilés, A., Sierra, J.L., Fernández-Valmayor, A. Building an Enhanced Syntax-Directed Processing Environment for XML Documents by Combining StAX and CUP. Fourth International Workshop on Flexible Database and Information System Technology. DEXA'09. 2009e.

68. [Sataluri 1998] Sataluri, S, R. Generalizing semantic rules of attribute grammars using loglc programs, Ph.D. thesis, Univ. of Iowa, Ames, Iowa. 1988.

69. [Schwerdfeger & Van-Wyk 2009] Schwerdfeger A. Van-Wyk Eric. Verifiable Parse Table Composition for Deterministic Parsing. 2nd International Conference on Software Language Engineering. 2009.

70. [Seroul 2008] Seroul, R., Levy, S., Foata, D. A Beginner's Book of TEX. Springer. 2008.

71. [Sierra et al. 2008b] Sierra, J.L., Fernández-Valmayor, A., Fernández-Manjón, B. From Documents to Applications Using Markup Languages. IEEE Software 25(2), 68-76. 2008b.

72. [Sleeman & Brown 1986] Sleeman, D., Brown,J.S (eds.). Intelligent Tutoring Systems. Academic Press. 1986.

73. [Stanchfield 2009] Stanchfield, S., ANTXR: Easy XML Parsing based on The ANLR Parser Generator. Java Due.com, Hillcrest Comm. & FGM, Inc. javadude.com/tools/antxr/index.html. 2009.

74. [Steams & Hunt 1985] Steams, E.R., Hunt III, H.B. On the Equivalence and Containmnet Problems for Unambigous Regular Expressions, Regular Grammars and Finite Automata. SIAM J. of Computing, 14(3), 598-611. 1985.

75. [Temprado et al. 2010a] Temprado, B., Sarasa, A., Sierra, J.L. Managing the Production and Evolution of e-Learning Tools with Attribute Grammars. The 10th IEEE International Conference on Advanced Learning Technologies. ICALT. 2010.

76. [Temprado et al. 2010b] Temprado, B., Sarasa, A., Sierra, J.L. Modular Specifications of XML Processing Tasks with Attribute Grammars defined on Multiple Syntactic Views. Fifth International Workshop on Flexible Database and Information Systems Technology. FlexDBIST. 2010.

77. [Vlist 2003] Vlist, E. Relax NG. O'Relly. 2003.

78. [Tomita 1986] TOMITA, M. Efficient Parsing for Natural Language. Kluwer Academic, Boston. 1986.

79. [Visser 1997] Visser, E. Syntax definition for language prototyping. Ph.D. thesis, University of Amsterdam. 1997.

80. [Xiaoqing et al. 2010] Xiaoqing Wu, Barrett R. Bryant, Jeff Gray, Marjan Mernik: Component-based LR parsing. Computer Languages, Systems & Structures 36(1): 16-33. 2010.

81. [Younger 1967] Younger, D. H. Recognition and Parsing of Context-free Languages in time n3. Information and Control 10(2): 189-208, 1967.